FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Workflow engine for parallel batch processing

João Oliveira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. João Pascoal Faria

February 20, 2017

Workflow engine for parallel batch processing

João Oliveira

Mestrado Integrado em Engenharia Informática e Computação

February 20, 2017

Abstract

In the context of a software platform that performs complex workflows to analyze SAF-T files (Standard Audit File for Tax Purposes) in batch mode, the need to impose complex restrictions to the sequencing and concurrency of each task arises. The purpose of this work is to identify relevant restrictions that may need to be imposed on workflows, as well as distributing and monitoring their execution among any number of "slave" machines, that perform the actual computational work of each task of the workflow. The final solution should improve both flexibility in workflow orchestration as well as performance improvements when running multiple workflows in parallel.

Besides analyzing the existing system and eliciting its requirements, a survey of existing solutions and technologies is made in order to architect the final solution. Although this work aims to improve the existing system from which it arose, it should be developed in an agnostic manner, so as to be integrated with any system that requires the handling of complex computational workflows.

In the end, a modular system was designed, implemented, tested and compared with the existing one. However, each layer was developed as a generic and re-usable system, and although together they aim to solve this particular problem, they can be used to solve different problems with similar needs. Therefore, each integrating part of the solution is detailed and validated, before integrating them all and comparing the result with the existing version of the platform.

The final solution was able to reduce the total processing time of multiple concurrent files, as well as adding the capability of distributing task processing, using any number of machines, allowing it to scale horizontally. Finally, more features were added to the file importation workflows, which now have a wider set of capabilities which can be used in the future to improve the product.

Keywords: Concurrent algorithms, Concurrency control, Layered systems

Acknowledgements

First I must thank João Pascoal Faria for supervising this thesis with such valuable inputs and for showing genuine care and enthusiasm for the result of this dissertation. He helped steer this work towards the proper direction and I have learned and grown quite a lot while working with him.

Secondly, I want to thank Valter Pinho for challenging me to develop this dissertation project at Petapilot and helping me achieve it since then. He told me many times that my goal of finishing this dissertation was also his own, which should say enough about is commitment to this cause.

However, not only Valter, but I want to acknowledge everyone who worked with me at Petapilot, for helping and teaching me so much, making me grow as both a person and a professional. However, a special mention must be made to Bruno Sousa and David Cunha who gave several inputs that helped this dissertation move forward.

I must also deeply thank my parents, Carlos and Teresa Oliveira to whom I owe not only this dissertation, but my entire education.

A special thank you to my girlfriend, Carla Santos, that constantly motivated me to do better and made me carry on during the toughest moments of this dissertation and my whole masters degree.

To all my close friends who have been with me for years and deeply contributed to who I am today: Diogo Pereira, Rui Pedro, João Meira, Ricardo Sousa, Miguel Germano, Mike Pinto, Mário Fidalgo, Carolina Valdoleiros, Bárbara Almeida and Sara Sousa.

Last, but not least to Tiago Moreira. Not only has he been a long term friend and a former colleague at this institution, he has also been a mentor at an academic, professional and personal level.

To all these people and to the entire staff at FEUP, particularly those involved in this masters degree: my sincerest thanks,

João Oliveira

"A rock pile ceases to be a rock pile the moment a single man contemplates it, bearing within him the image of a cathedral."

- Antoine de Saint-Exupéry

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
JVM	Java Virtual Machine
KPI	Key Performance Indicator
SAF-T	Standard Audit File for Tax Purposes

Chapter 1

Introduction

This work is originated from the needs of a platform that extracts and validates data from different types of files, through the definition of workflows, composed by a sequence of tasks that must be executed to process a file. However, the system currently has some limitations in both scalability for lack of distribution options (i.e. currently the workflow must be managed and executed in the same machine) and flexibility in terms of allowing more complex workflows with different ways to create paths between tasks. In order to better understand the requirements of this workflow engine, this chapter will detail the current system and its needs, giving some context and reasoning to each of the features the engine should support. However, it is important to keep in mind that, although this solution aims to fulfill the needs of an existing system, it should be designed and implemented in an agnostic way, so that it can be used for other types of workflows that can take advantage of theses features.

1.1 Current system - Colbi Audit

Colbi is a web platform created at Petapilot to automatically process analytical contents and provide information about a company. This is achieved using raw data from various sources, but particularly from SAF-T (Standard Audit File for Tax Purposes) files[fDoBR17]. These come in several variations (e.g. SAFT-T PT, SAF-T LT, FAIA), each with its own structure and contents. The system that handles these workflows and processes the files is called *Colbi's "Core"*. Besides storing the raw data in a repository, several integrity or structural rules are checked both at the level of that particular file and the entire company. Some KPI (key performance indicators) are also calculated as well as other data transforming operations. To achieve this, each file type has a *workflow* associated to it, which is currently defined in YAML files as shown in listing 1.1[Yam17]. When a user uploads a SAF-T file to the system, an initial sensing is made to determine what type of file was introduced, so that the proper workflow may be initiated. Each *workflow* represents a

```yaml
name: import_workflow
tasks:
  - name: parse
  - task: Parse
  events:
    success:
      - name: merge
        task: Merge
        parameters:
          adapter: mysql
        events:
          success:
            - name: calculate_kpi
              task: KPI
            - name: calculate_trial_balance
              task: TrialBalance
```

Listing 1.1: An example of a workflow definition in the current system, using YAML notation

sequence of tasks. A task represents a single step in the importation of a file, and can be viewed as an atomic unit of work or processing that must be done.

Note that tasks can receive parameters, much like functions, in two distinct ways. The first one is in the workflow definition itself, and as such, the value will always be the same for that particular workflow type. The second one is through a set of workflow parameters that are kept throughout the entirety of a workflow execution. Tasks can write and read from these parameters, which are essentially a map of string identifiers to string values. This way, tasks can communicate with each other, passing along information that is discovered in runtime.

During the workflow (usually at a very early stage), two important characteristics are identified: the fiscal period and the Tax Registration Number to which the data contained in that file refers to. These are particularly important because conflicts may arise when executing *workflows* with both these values in common. When this happens, there are a particular set of scheduling rules that must be supported, both inside a *workflow* and while executing multiple *workflows* at the same time. In order to better understand these scheduling rules, a few example tasks, which are shown in listing 1.1 should be introduced:

- The *Parse* task is responsible for reading the contents and extracting the data contained in the SAF-T files.
- On the other hand, the *Merge* task detects identical entities in the data repository (e.g. two identical invoices) and selects only one of them to be considered during others steps of the workflow, or the visualization of data itself.
- The *KPI* task computes an assortment of *Key Performance Indicators* (e.g. total sales amount, number of distinct clients)

- The *Trial Balance* task, as the name indicates, calculates a company's Trial Balance.

This example allows us to understand the scheduling restrictions the current system can impose on tasks:

1. *Sequentiality* - Some tasks depend on the previous execution of other tasks and can only execute after those tasks are successfully completed. For example, the *KPI task* can only execute after *Merge*, otherwise it may consider duplicate data and produce incorrect values.

2. *Parallelization* - On the other hand, some tasks can be executed simultaneously, since they do not interfere with each other. Both the *KPI* and *Trial Balance* tasks only read tables, storing their result in separate locations. Therefore, they can be executed at the same time without interfering with each other.

3. *Alternative* - Depending on the success or failure of certain tasks, the *workflow* may choose to run a different set of tasks.

Currently, the existing system is responsible for determining when tasks can be executed, as well as the execution itself. This makes it impossible to distribute task execution, which is restricted to a single machine.

1.2 Improvement needs

The current system could be improved with a workflow engine with a larger set of features, both in terms of task scheduling and work distribution, to achieve greater performance and customization. This section describes the identified requirements that could benefit the existing system.

1.2.1 Scheduling rules

As explained before, in terms of scheduling rules, the current system supports both sequentiality and parallelization, which should be maintained. However, alternative could be improved to take into consideration the result of a task, independently of its type. For example, the task could return an integer result and the workflow would follow a different execution path according to that result.

When running multiple *workflows* simultaneously, some tasks may be executed only once, instead of one time per *workflow*, while yielding the same result. These tasks usually have long execution times, making it valuable to wait until all running *workflows* are ready to execute them. A prime example of this is the *Merge task*. Besides being a long task, particularly when the size of data starts to grow, the results of running it twice on the same company and fiscal year are exactly the same. Therefore, when multiple *workflows* over the same data are being executed, it would be worthwhile to wait until all of them are ready to run the task and then execute it only once. A suitable designation for this scheduling rule would be "grouping", since tasks create groups which will share a single execution.

1.2.2 Work distribution

The current system is simultaneously responsible for managing the workflows and executing each task. This *ad hoc* architecture introduces some limitation in terms of distributing the task executions throughout multiple workers. A worker can be a machine or a thread, essentially representing an agent that knows how to execute tasks. This way, one of the system requirements is that tasks can be distributed among an arbitrary number of workers. These can be all in the same machine or spread out among several. In order to maximize parallelization, the *workflow* orchestration should strive to maintain the maximum possible number of workers occupied at any given time.

1.2.3 Resource locking and prioritization

In some cases, some tasks may require access to certain resources, while preventing other tasks to access it at the same time. Usually, the operation itself prevents this, in cases such as writing to a relational database table, which will become locked by the database engine itself. However, this means that the task execution will be paused until the resource it needs becomes free again. While this happens, a worker will be considering being busy, although doing no actual computations at all. To prevent this, each task may also request locks, imposing some rules on processes that can run simultaneously, which we can then manage before assigning tasks to workers. Locks are simple string identifiers referring to a certain resource (e.g. a file or a database table). These can be of one of two types: shared or exclusive. Tasks with exclusive locks can not be executed simultaneously with any other task requesting the same lock, be it of any type. On the other hand, shared locks can run simultaneously with each other, but not with exclusive locks.

Additionally, some tasks could have more priority over others, in the sense that they should finish earlier. For example, we may want to import a certain file as quickly as possible, making its tasks have more priority. Thus, the system should attempt to run higher priority tasks first, but without breaking the principle of maximizing parallelization.

1.2.4 List of requirements

For the purpose of better readability and consolidation, the following is a list of the requirements the workflow engine must support.

R1: Sequentiality Enforce that a certain task may only be executed after another certain task is completed.

R2: Parallelization Specify that certain tasks can run simultaneously. The system should attempt to do so, when possible.

R3: Alternative Depending on the result of a certain task, the system should be able to select the next task accordingly, if such a condition is requested in the workflow definition.

R4: Grouping Enforce that when multiple workflows are executing simultaneously, tasks marked as groupable should be executed only once and when all workflows are ready to do so.

R5: Work distribution The system should be able to distribute tasks among an arbitrary number of workers, which can be deployed for example in remote machines or multiple threads in the same machine.

R6: Resource locking A task may request shared or exclusive locks in the form of a string identifier. The system must guarantee that no tasks are running simultaneously without assuring these locks.

R7: Prioritization A task may have a certain priority value. The system should attempt to perform higher priority tasks first, while still striving to maintain workers as occupied as possible.

1.3 Goals

In summary, the goals of this work are the following:

- Improve the flexibility of the existing system by adding more features in terms of workflow execution, through requirements *R1*, *R2*, *R3*, *R4* and *R7*;
- Increase performance while processing files concurrently, even if at the cost of the processing of a single file becoming slower, through requirements *R4*, *R5* and *R6*;
- Create a generic set of modules that can then be used, together or separately by other systems with similar workflow processing needs. Although the main purpose is to solve the problems and limitations of *Colbi Audit*, it is more valuable to create customizable and reusable software, which can then be reused by different client applications.

1.4 Document structure

This document will first present a literature review on chapter 2, as well as an overview and architecture of the envisioned solution on chapter 3. After that each layer of the architecture will be explained in detail in its own chapter, with chapter 4 detailing the workflow orchestrator, chapter 5 detailing the locks and prioritization service and chapter 6 detailing the workers engine.. The integration with the existing system is then described in chapter 7, including a comparison between the existing and the developed system. This in turn will lead to conclusions and identification of future work in chapter 8.

Chapter 2

Literature review

In order to conceive and implement a system supporting the requirements mentioned in 1.2.4, an analysis of current systems and literature should be made. This can help by finding existing systems that can be directly used or extended and by learning from other systems that implement the same or similar features.

2.1 Workflow management

2.1.1 Petri nets

Petri nets are presented by [Mur89] as a graphical and mathematical modeling tool that allows the description and studying of information processing systems characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic. He also explains that there is only one rule to learn about Petri nets, which is related to transition enabling and firing. The Petri net itself is a directed, weighted, bipartite graph, which may contain two kinds of nodes: places and transitions. Arcs can only exist between nodes of different types, and their weight represents a set of parallel arcs (i.e. a weight of 5, represents 5 parallel arcs). Finally, a marking, which represents a state of the Petri net, assigns to each place a number of tokens. With these concepts in mind, the rules for transition enabling and firing can now be understood, as explained by Murata:

1. A transition is enabled if each input place contains a number of tokens equal or greater to the weight of their transition arc.

2. An enabled transition fires when the event associated with it happens.

3. When an enabled transition fires, the number of tokens required from the transition are moved from each input place to each output place of the transition, according to the weights of the connecting arcs.

An application of Petri Nets to Workflow management is explored by [Van98]. In his work, he states some of the points in favor of Petri nets applicability in this domain:

- Formal semantics - a workflow process specified by a Petri net has a clear and precise definition, due to the formal definition of its semantics, as well as of its enhancements (color, time, hierarchy).
- Graphical nature - Petri nets can be represented graphically, which makes them easy to learn and supports communication with end-users.
- Expressiveness - the author states that Petri nets support all the primitives needed to model a workflow process. However, whether if holds true for the particular case of this work must be evaluated further.
- Properties - There is a vast amount of investigation in the properties of Petri nets, resulting in a lot of common knowledge.
- Analysis - There are several analysis techniques which allows the assessment of metrics and assurance of properties.
- Vendor independent - Petri nets are not associated to any specific software vendor, being a tool-independent framework.

To explore the applicability of Petri Nets as a modeling tool for this system, we can attempt to represent each of the proposed scheduling rules. *Sequentiality*, *Parallelization* and *Alternative* could be achieved using the routing constructs proposed. However, modeling other constructs such as *Grouping* or resource locking becomes cumbersome and requires extensions to be applied on top of the standard Petri Nets[He04].

2.1.2 Workflow execution engines

Analyzing the current workflow execution engines in the market can help us perceive if there is any existing system that can be used. On the other hand, if no such system exists, we can still learn from other systems by understanding how they implement features similar to our requirements. This way, the following systems were analyzed:

Apache Taverna is an open-source tool for workflow design and execution. It features a graphical interface for designing the workflows. However, it is heavily oriented towards workflows with manual tasks, being mainly oriented towards scientific workflows[Tav16].

YAWL Yet Another Workflow Language also features a graphical interface for developing workflows, oriented for both manual and automatic tasks, supporting integration with external web services It supports a wide range of scheduling rules, but still does not support grouping[Fou16].

WDL Workflow Description Language is an open-source project that attempts to create an human-friendly language for workflow definition. Although it has limited scheduling features, it could be interesting to learn or even extend it for the workflow definition of our system[Lan16].

Triana Also open-source, this engine is more oriented towards problem solving which is particularly good at automating repetitive tasks. Although it also features a graphical tool, it supports the creation of workflows programmatically in Java[Tri16].

Pegasus Oriented towards large-scale workflow managing, Pegasus is interesting for being the only workflow engine analyzed which supports the concept of remote execution in a distributed manner, similarly to the worker distribution required by our system[DVJ+15].

Table 2.1: Existing workflow engines and their features

Engine	R1. Sequentiality	R2. Parallelization	R3. Alternative	R4. Grouping	R6. Resource locking	R7. Priority	R5. Worker distribution
Apache Taverna	Yes	Yes	Yes	No	No	No	No
YAWL	Yes	Yes	Yes	No	No	No	No
WDL	Yes	Yes	Yes	No	No	No	No
Triana	Yes	Yes	Yes	No	No	No	No
Pegasus	Yes	Yes	Yes	No	Yes	No	No

Table 2.1 displays the workflow engines analyzed and which features of the system requirements they support, which noticeably are the same across all entries. There are several reasons for this:

- Workflow engines usually predict the existence of manual tasks that require human interaction that may take an indefinite amount of time. The required engine supports fully automatic workflows.
- The analyzed engines only take into consideration the execution of a single isolated workflow. In this specific scenario, multiple workflow instances will be executed simultaneously, with interactions between them, particularly with the grouping feature.
- In terms of resource locking, usually it is expected that the tasks themselves take care of it, so engines do not perform preemptive locking.

Note that although some features are shown here as non-existing, this is only in the context of workflow execution engines. There are other systems that have these capabilities, albeit in a different context.

2.2 Scheduling and concurrency control

One type of system that faces similar concurrency issues are databases. Relational databases face similar concurrency problems when reading and writing to tables. Usually, these operations are organized within transactions, which transform consistent database states into another consistent state. In practice, these transactions are not executed in isolation, but concurrently with other transactions, much like our workflow tasks. As such, they can be organized in *schedules* which are

the sequence in which the operations of the transactions occur. One way to schedule transactions is to perform all operations of a transaction, followed by all the operations in the next transaction and so on. This is called a *serial schedule*. On the other hand, there could be other possible schedules, whose results would be equivalent to those of the serial schedule. These are called *serializable schedules*. Although they intertwine the operations of the transactions, their results are the same as performing them sequentially. However, there is an even stricter condition called *conflict-serializability*, which takes into account that there may be some operations in which changing their order alters the behavior of at least one of them.

In order to create schedules that obey these restraints, relational databases also employ locking mechanisms. One of them is called *Two-Phase Locking* (2PL), which, as the name indicates, is composed of two phases, one where locks can only be acquired and another where locks are released. However, supporting different types of locks can allow for a more efficient lock management. For example, when reading a value, the system does not need to lock other reading operations. As such, similarly to the requirements of our workflow engine, databases can use shared and exclusive locks. Due to the similarities of both systems, it is worthwhile to understand how databases implement this particular locking mechanisms.

Garcia-Molina *et al.* propose a way to keep track of locks by inserting locking actions into the schedule. These actions help keeping track of what is locked by updating a lock table. This table maps database elements to information regarding the locks requested on that element. If the database contains no entry for a certain element, we can immediately tell it is unlocked[GMUW08].

2.3 Technology overview

Although the envisioned system aims to provide a generic service for workflow management, the existing system with which it will be integrated and validated with, as well as the context in which that system is developed, must be taken into account when selecting the technology with which the system will be implemented. Easy integration with the existing system is crucial in order to guarantee interoperability and reduce implementation effort. Since Colbi's *Core* is entirely implemented in *Java 7* the selected language should, at the very least, run on the *JVM*, preferably with *Java* interoperability. Concurrency should be a recurring problem during the implementation, so the selected language and/or libraries should facilitate solving such problems. Finally, this should be production-ready software, so experimental or unstable components should be avoided.

2.3.1 Scala

The selected language for implementing the various software modules was *Scala*, for various reasons:

1. It is both functional and object oriented, since everything is an object (including functions) [OAC+04], which may be bring advantages when dealing with concurrency problems.

2. The functional paradigm also makes dealing with collections much easier, which is something that was only introduced in *Java 8*, through lambda expressions;

3. It is a *JVM* based language, which can be used together with *Java*, meaning that *Java* classes can be imported and used in *Scala* classes and vice-versa. This allows the reutilization of existing code if needed, as well as using the new modules in existing code;

4. There was interest at the company to experiment with the language and determine if it would be suitable for new developments. Since new software modules were being created from, it was deemed an interesting opportunity to do so;

2.3.2 Akka framework

The *Akka* framework was used in different parts of the implementation for various reasons which are explained when detailing the modules it was used in. Essentially, it is an implementation of the Actors model[Agh86], which is aimed at concurrent and distributed scenarios. It features implementations in both *Java* and *Scala*, thus fitting in the technology already used by the current system.

What follows is a brief summary of the main *Akka* features used throughout the implementation, which are important to understand how the system works.

2.3.2.1 Actors model

An actor has a behavior which defines a way to interpret incoming messages, which can trigger the following events:

1. new actors may be created;

2. the actor may change its behavior, thus replying differently to messages henceforth;

3. messages may be sent to other actors.

Note that these events don't require a particular order. A set of actors, the way they relate with each other (trough supervision which will later be explained) and the messages awaiting delivery all form an *Actor System*. This model will be used throughout this implementation in various concurrency scenarios, with its advantages being discussed throughout this document, while the implementation itself is described.

2.3.2.2 Creating actors

To create an actor, the client application needs to first create a class which integrates the *Actor* trait provided by *Akka*. When doing so, this class must implement the *receive* method, which represents the default behavior of an *Actor*. Inside this method, pattern matching can be used to determine how the actor reacts to that message. The following code snippet shows an example of an Actor implementation:

```
object ExampleActor {
  case object Message1
  case object Message2
}
class ExampleActor extends Actor{

  override def receive: Receive = {
    case ExampleActor.Message1 =>
      println("I␣have␣received␣message␣1!")
    case ExampleActor.Message2 =>
      println("I␣have␣received␣message␣2!")
  }
}
```

This way, actors of this type can be created either inside other actors or by directly using the instance of an *ActorSystem* provided by *Akka* .

2.3.2.3 Message sending

Messages can be any object, however, the objects sent should be immutable, otherwise different actors can change the message, interfering with each other. *Akka* offers two different methods to send messages. The first and most common one sends a message in a "fire-and-forget" manner, meaning that the message is sent asynchronously and the method immediately returns. The sender will not block and may continue executing other tasks, but it will never know when the receiver got the message. This is known as *tell* and has an exclamation point as a convenient alias.

```
actor ! Actor.ExampleMessage
```

Listing 2.1: An example of a *tell* message in *Akka*

The other method is used when the sender expects a response from the receiver and is thus called *ask*, using a question mark as an alias.

```
val future = actor ? Actor.ExampleMessage

future onComplete {
  foo()
}
```

Listing 2.2: An example of a *ask* message in *Akka*

Although the message is still sent asynchronously, this method returns a *Future* representing a possible reply from the receiver. Either way, the order of delivery of messages in only guaranteed on a per-sender basis i.e. , messages from the same sender will arrive at the same order they were sent, but the same may not happen for messages of different senders.

Table 2.2: The dispatchers provided by the *Akka* framework

	Description	**Thread distribution**	**Mailboxes**
Dispatcher	Default event-based dispatcher	Assigns a set of actors to thread pool	One per actor
PinnedDispatcher	Each actor has a thread pool with a single thread	One per actor	One per actor
BalancingDispatcher	Tries to balance work between the actors All actors should now how to answer messages	Balanced	Single shared mailbox
CallingThreadDispatcher	Runs only on the calling thread Usually only for testing purposes	Only the calling thread	One per actor

2.3.2.4 Changing behavior

Akka also provides a way for actors to change their behavior. To achieve this, the user application may implement other methods like the default *receive* it is forced to implement. Then, at any point inside the actor, it can invoke the *become* method, which receives the implemented function as an argument. Note this is possible because functions in *Scala* are also objects. From that point on, messages received by that actor are handled by that method. Additionally, the actor can use the *unbecome* method to revert to its last behavior.

2.3.2.5 Supervision

Inside an actor system, *Akka* organizes each actor in an hierarchical manner. This means that all actors are supervised by some other actor. When an actor creates another, it will automatically be its supervisor. Even actors directly created in the actor system (i.e. "root" actors) have a special supervisor which is created and managed by *Akka* itself. Supervision means that, in case of failure, the actor will suspend all its children and warn its supervisor. Each actor can then implement a supervision strategy which can handle each cause of failure (i.e. a *Throwable* object) separately. To better understand this, we can compare this to a worker running inside a *try* statement with the supervision strategy being its respective *catch*. Besides implementing some contingency logic, the supervisor can either *a*) resume the actor, keeping the same instance of the actor running, *b*) restart the actor, which will essentially destroy the existing actor and create a new instance, *c*) stop the actor completely, or *d*) escalate the failure to its own supervisor, thus failing itself.

2.3.2.6 Dispatchers

Every actor system has a *Dispatcher* which essentially handles the execution of the actors, determining how threads are given to workers to handle messages and how mailboxes (which hold the messages for actors) are assigned. They also determine how threads can be shared by actors. It is important that a proper dispatcher is selected, according to the needs of the client application. Table 2.2 details the dispatchers provided by *Akka* . Note that these dispatchers can be configured to some degree, but these configurations will not be discussed in this document.

2.4 Conclusions

Although the performed analysis found no system that could be used to help in the implementation, it provided a solid ground, upon which a solution can be envisioned. No existing workflow engine analyzed was found to fulfill all the system requirements, but this analysis brought some insight about how workflows are usually modeled and how the main scheduling rules are handled. During the implementation of the final system, techniques from these existing systems could be useful to learn from. The analysis of scheduling and concurrency control in relational databases also introduced some important concepts and techniques that are used in many production systems to handle situations much like the ones this workflow system will face to ensure the same restrictions. With this in mind, the novelty of this system comes from the simultaneous support of the mentioned features, particularly the fact that it takes into account the concurrent execution of multiple workflows, allowing interaction between them (e.g. the grouping feature).

Chapter 3

System architecture

In order to achieve the mentioned requirements, a modular architecture was envisioned, with each module having a particular set of responsibilities:

3.1 System components

Workflow orchestrator: responsible for ensuring that the scheduling rules are maintained (requirements *R1*, *R2*, *R3* and *R4*), dispatching tasks to the next module as they become available;

Priorities and locking handler: called *NoQueue*, prioritizes tasks and ensures resource locking, sending them to the final module in priority order, as long as they are not locked (requirements *R6*, *R7*);

Workers engine: maintains a pool of workers which retrieve tasks from *NoQueue*, executes them and sends the result back. The number of workers should be configurable and the workers should be distributable across different machines (requirement *R5*).

The lifetime of a task would be the following, as shown in figure 3.1:

3.2 Components interaction

1. When the file begins its importation, the workflow orchestrator determines the respective workflow and starts its execution.

2. When the task has no scheduling rules preventing it from running, it is sent to *NoQueue*. Here, the task is placed in a priority queue, working together with a semaphore pool to ensure that requested locks are respected.

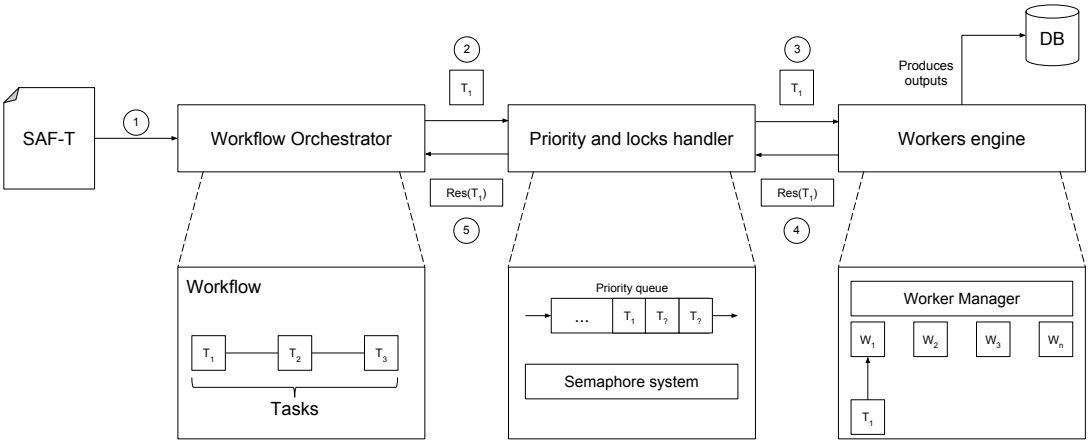

Figure 3.1: Proposed system architecture

3. Once a worker is available, the task with highest priority that is not locked is sent to that worker.

4. The worker executes the task, while being supervised by a manager, which ensures that the worker recovers from failures and properly reports them back.

5. When the task is finished, its result is reported back to the priority and locks handler.

6. Finally, the result of the task is sent to the orchestrator, which will then make appropriate decisions according to that result.

In practice, *NoQueue* acts as an intermediary between the orchestrator and workers, with some logic to select what tasks are handled to workers. As such, *NoQueue* must provide an interface that the orchestrator can use to insert and check the status of tasks, and workers to retrieve tasks and report their execution status. To achieve this, *NoQueue* will provide an HTTP API. On one hand, the orchestrator can perform a request to insert a task and then periodically perform a request inquiring about its status, until it is complete. On the other hand, workers can periodically perform a request asking for tasks to execute, as well as another request to notify *NoQueue* of the task execution result. This way, these three modules can run in the same or different machines, with the only requirement being that both the orchestrator and the workers know the address in which *NoQueue* is serving its API. Additionally, creating multiple instances of the workers retrieving tasks from the same instance of *NoQueue* is only a matter of having those instances performing requests to the same address.

The way these components communicate and how they handle their responsibilities internally will be described in more detail throughout this document.

Although the separation of these modules will create some overhead due to the communication that needs to be performed between each module, it allows for a simpler implementation with distributable components (i.e. each component can run on a different machine than the others).

Also, if these components are implemented in a generic way, they can be re-used separately for different use cases with similar needs.

Chapter 4

Workflow orchestrator

The workflow orchestrator provides a library written in *Scala* that client applications can use to define a sequence of tasks, with a set of properties that determine when they can be executed. The orchestrator will then automatically keep track of tasks and their results (which can be any object), sending them through a dispatcher when they are ready. A workflow is complete once all tasks have either been executed or are no longer reachable.

The main way to control the flow of tasks is by defining their transitions. A task can contain several transitions which are composed of a destination task and an expected result. It is considered that a transition is successful if the source task (the task that contains the transition) yielded the expected result for that transition. The orchestrator will use these transitions to determine what tasks can be executed (and dispatch them) and to determine which tasks will no longer be reachable, because their transitions can never be successful.

The orchestrator also allows client applications to request that tasks be grouped. What this means is that the orchestrator will wait until all similar tasks are ready to execute. Once they are, only one of the tasks will be executed and all the tasks will be considered completed with the yielded result. This makes it possible to group tasks in which the result of a single execution is the same as multiple sequential executions, thus avoiding unnecessary executions.

Tasks are defined by:

Identifier: an unique identifier which the client application should generate;

Name: a name for the task. This should be interpreted similarly to the name of a function, identifying somehow what the task should perform once it is sent to the dispatcher;

Parameters: a key-value set which can be used to send information to the dispatcher. If the name of the task is the name of the function, these are the arguments;

Groupable: whether the task can be grouped or not. This concept is explained further below;

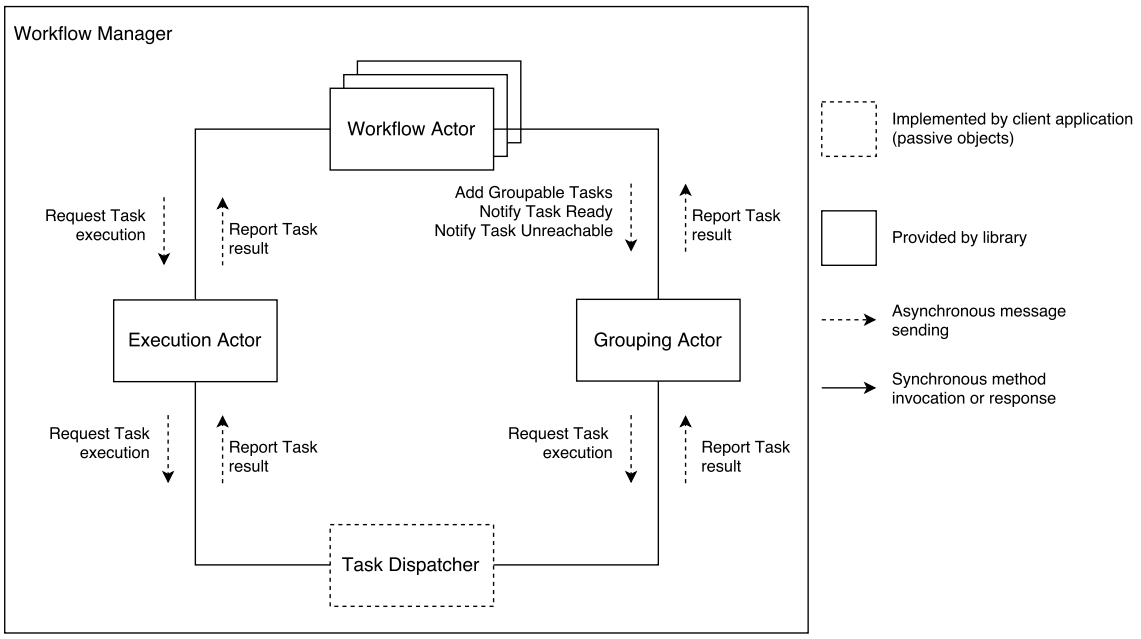

Figure 4.1: Conceptual model for workflow and task definition

Grouping parameters: a sequence of keys of parameters which values will be considered when checking if two tasks are similar and can be grouped. Can be empty;

Result: an initially empty value that should be overwritten by the task dispatcher once the task is done;

Transitions: the set of transitions which have this task as a source, each one represented by a pair containing the destination task and the expected result.

So, in order to group tasks, a grouping code is created to check if tasks belong to the same group or not. The code contains the name of the task, followed by the values of each parameter in the grouping parameters sequence. This is so that the client application can state which workflow parameters of the task must be the same in order for the task to be groupable, adding more customization to how the grouping is done, allowing to narrow it down to tasks with similar contexts.

With this model, formalized in figure 4.3, it is possible to create all the routing rules required by *Colbi*. For the purpose of the following examples let's consider tasks A, B and C which return the boolean value *true* when they are successful or *false* otherwise.

Sequence To make task A sequential to task B, we can simply add a transition from A to B when the result is *true*.

Parallelization To make tasks B and C parallel, we can add transitions from A to B and to C, both expecting *true*.

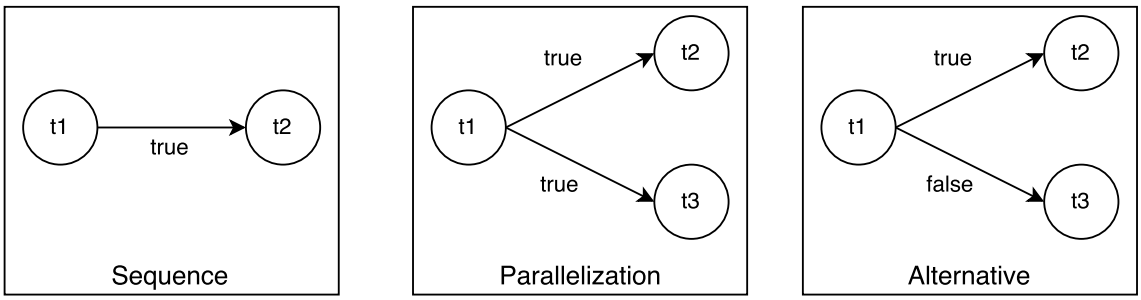

Figure 4.2: Examples of workflows for each required scheduling rule

Alternative If we want to execute *B* or *C* execute depending on the result of *A* we can add a transition from *A* to *B* with *true* as the expected result, and a transition from *A* to *C* expecting *false* as the expected result.

Figure 4.2 shows a graphical representation of the described workflows. Finally, a workflow can have a set of parameters (key-value pairs) which are specific to that workflow instance and are accessible by tasks, which can read from or write values to these parameters. The purpose of these workflow-level parameters is to provide tasks with a way to easily share a global context with each other and exchange information that may or may not alter their behavior. The client application can also implement a task dispatcher which should implement the required logic to either execute the task or send it to an external entity capable of doing so.

This chapter will first describe how the orchestrator is designed and some implementation details. Afterwards, it will present how client applications can make use of the provided API. Finally, it will detail how the library was tested and validated.

4.1 Design and implementation

The workflow orchestrator was implemented with an actors model, using the capabilities of the *Akka* framework. Since there can be multiple workflows simultaneously being executed, the actors model facilitates an implementation where specialized actors handle their responsibilities concurrently while exchanging messages that make the entire process, while guaranteeing the safety of data for which each actor is responsible. This way, isolation between each workflow instance is easier to maintain, as well as concurrency without worrying about synchronized methods or locking access to variables.

4.1.1 Running workflows

Before going into any detail on how the workflow orchestrator was implemented, it is crucial to understand how a single workflow is expected to work. For now, the problem of grouping tasks will not be discussed, as it will be controlled by a component external to the workflow, as will be explained later in this section. In order to better grasp the concepts discussed, it is helpful to

think of workflows as a directed labeled graph[Wil86], where tasks are represented by nodes and transitions as edges with their expected result as a label.

When the workflow starts, every task with no incoming transitions is considered ready to execute and dispatched to the retriever. Note that if there are no tasks under this condition, no task in the workflow will be able to execute.

On the other hand, when a task ends, a new set of tasks may become runnable. However, instead of analyzing the entire workflow, we can first narrow it down to the tasks with incoming transitions that have the finished task as the source. A task is considered ready if, for every incoming transition, the source task was completed and yielded the result required by that transition. On the other hand, if there is at least one incoming transition where the task has a different result or is unreachable, the destination task is also considered unreachable. So, when a task ends, we can immediately mark all destination tasks on outgoing transitions with different results as unreachable. After that, we can continue traversing the graph, marking all visited tasks as unreachable. Marking tasks as unreachable is useful because we can immediately discard them and avoid having to repeatedly calculate if it is runnable or not. Additionally, the workflow has to keep track of unreachable tasks to understand if it is finished or not, which should happen once all tasks are either completed or unreachable. However, discarding unreachable tasks this way will make it impossible for alternative branches to merge back into a single execution path, since one or more of the alternative tasks will be considered unreachable, along with any task from outgoing transitions.

4.1.2 Workflow manager

Client applications will mainly will instantiate and interact exclusively with the workflow manager. This object is responsible for determining when tasks can be run and send them to the dispatcher, which the client application must provide in order to create a worker manager instance. Note that client applications are free to create as many instances of workflow managers as they want, but workflows will only group tasks within the same instance of a workflow manager.

The workflow manager contains the following objects, also shown in figure 4.3:

Workflow Actors: for each workflow created in the manager, it will create an instance of a workflow actor that will be responsible of handling that particular workflow.

Execution Actor: this actor will receive tasks to be executed from the workflow actors. It should send those tasks to the dispatcher and monitor their execution. Once they are done, it must notify the workflow actor that requested the task.

Grouping Actor: workflow actors will first warn this actor of the tasks they will want to group, along with their grouping code. While the workflows are executing, they should also warn this actor once grouping tasks are ready or unreachable. This actor will use this information to determine when to run one of the tasks of the group by sending it to the task dispatcher, sending back the result to the workflows from which the tasks of the group belong.

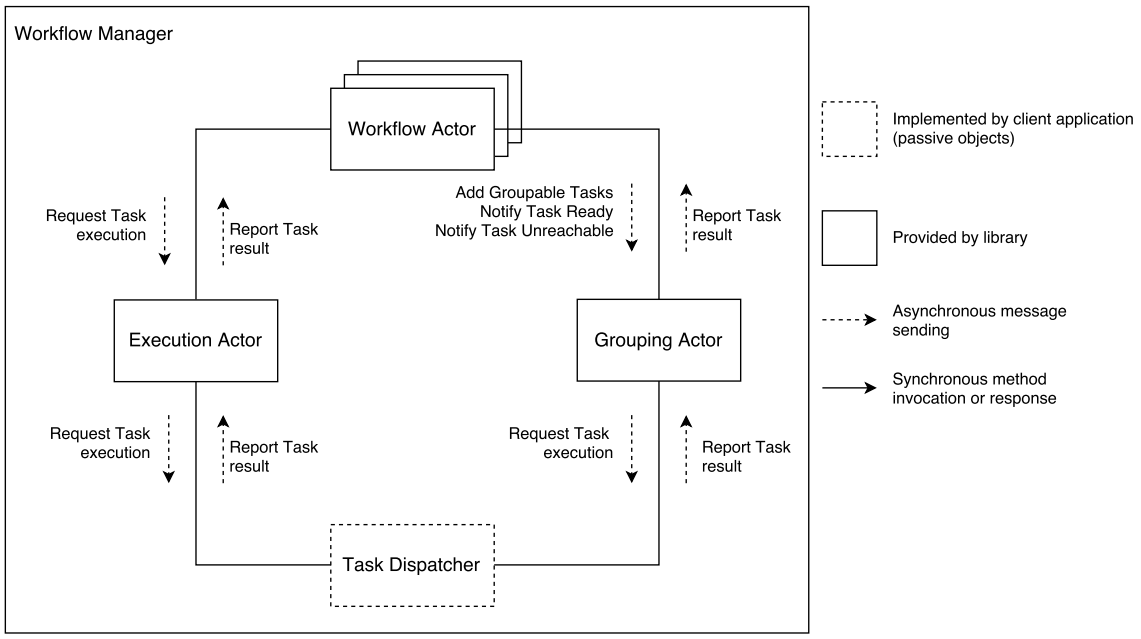

Figure 4.3: Workflow manager objects and their interactions

Task dispatcher: implementation provided by the client application that contains logic related to execute a task. This is a passive object whose methods will be invoked by other objects, without doing anything by itself.

Upon being created, the execution and grouping actors will automatically be created. However, workflow actors are only created as workflows are inserted into the manager.

4.1.3 Workflow actors

As already mentioned, each workflow instance will have an actor responsible for properly sending tasks to be dispatched, as they are ready to execute throughout time. Internally, the actor will place a task on one of four collections: tasks waiting to run, tasks currently running, tasks that were already finished and tasks that are unreachable. Initially, all tasks will be considered waiting.

When the actor receives a message to start, it must first send a message to the grouping actor containing all the groupable tasks. After doing this, it can now take all tasks with no incoming transitions, move them to the running collection. These tasks are then split into two groups, based on whether they are groupable or not. Groupable tasks are sent in a message to the grouping actor, warning it that these tasks are ready to run. The remaining tasks are sent to the execution actor. Both will take care of executing the task and warning their respective workflow actor of its completion, although in different ways, which will be detailed later.

Note that before sending tasks to be executed, the actor will write the current workflow parameters to the task's internal parameters. In practice, a task will be executed with the workflow parameters as they were when the task was sent. Eventually, the workflow actor will receive a

message notifying it that a task is completed with its respective result. When this happens, besides updating the task's result, it will merge the workflow parameters of the task with its own instance of the workflow parameters. New values will be simply added to the parameters, while existing ones will be overwritten by the task. The task is then moved from the running tasks to done tasks. Next, tasks that are now unreachable will be marked as such and the grouping actor will be notified of this happening. This is important, because otherwise the grouping actor would be waiting for an unreachable task to become ready, which never happens. To determine unreachable tasks, it marks all destination tasks of unsuccessful tasks as unreachable. As a task is marked as unreachable, all its transitions' destinations are also marked in a recursive method that will mark all that path as unreachable. The remaining process is similar to when the workflow starts, but instead of considering tasks with no incoming transitions, it needs to determine which tasks have become runnable. To do so, it goes through its successful transitions and evaluates if the destination task has become runnable (i.e. all its incoming transitions are successful). Again, these tasks are either sent to the grouping or execution actors, accordingly.

As workflows are finished, a garbage collector will periodically delete them from memory to avoid indefinitely growing in terms of memory usage.

4.1.4 Task dispatching

As already explained, the client application must provide a way for the workflow orchestrator to dispatch tasks by either executing them locally or sending them to external agents to be executed. To achieve this, the workflow manager will need to receive a *WorkflowTaskDispatcher* object, which is an abstract class. So, client applications will have to extend this class by implementing two methods shown in listing 4.1.

```
/**
  * Callback invoked when a workflow finishes
  * @param id Name of the workflow that finished
  */
def workflowFinished(id: String): Unit

/**
  * When a task is ready to execute, this method will be invoked
  * The necessary steps to run the task should be implemented here
  * @param task The task that is ready to run
  * @return
  */
def dispatchTask(task: WorkflowTask): Any
```

Listing 4.1: Methods task dispatchers must implement

Internally, the dispatcher will invoke the *dispatchTask* method inside a *future*, so that the actor that requests the task execution is not blocked. When the future is complete, the dispatcher will send a message to the actor that requested the task, warning it of the task completion. This dispatcher will be used by both the execution and grouping actors.

4.1.4.1 Execution actor

Upon receiving a message to execute tasks, this actor will first store in a map the requested tasks and which actor requested it. Then, it will request the dispatcher to dispatch the task, using a method it provides that will invoke the *dispatchTask* method inside a *future*. Eventually, it will receive a message telling it that the task is complete, which will simply be forwarded to the actor that requested the task.

4.1.4.2 Grouping actor

The grouping actor is similar to the execution actor in the sense that it is also used by the workflows to dispatch tasks, with the difference that it may not dispatch tasks as soon as they are requested. Instead this actor will group tasks according to their grouping code. This code is a string generated for each groupable task, which always contains at least the name. However, if grouping parameters are specified, their values will be appended to this string. Since the grouping parameters are stored in a sequence, the order in which they are defined is relevant, because that is the order in which they will be appended to the grouping code preceded by a '-' character. Client applications must take this into consideration when using grouping parameters. To make this clearer, consider the following example: a workflow which retrieves products from different sources, has a task called "average" which calculates the average price of the products that were extracted, storing the value in a database table. While it is acceptable to repeatedly calculate the average value every time the workflow reaches that task, only the last value prevails, so it is better to group this task and just do it once, so the client application can activate grouping on this task to achieve this. All these tasks will have the "average" grouping code and only one of them will be executed once. However, we now want to separate our products of different categories in different databases, so we only want to group tasks that will calculate the average of a single category. The client application can then add the "category" workflow parameter to the grouping code. In this case, the grouping codes of the tasks will be, for example, "average-electronics" or "average-clothing".

When a workflow starts, it will send a message to the grouping actor to subscribe tasks. These tasks are then distributed to their appropriate groups, creating them if they don't exist yet. Each group contains a collection of waiting tasks and another of ready tasks. As tasks become ready, the workflow actors will also notify the grouping actor. Inside their group, the tasks will be moved from running to ready. Once all tasks on a group are ready, one of them is selected and sent to the task dispatcher. After that task is executed, a notification is sent to the workflow actor that requested each task, informing it of the completion of that task with the result returned by the task that executed.

4.2 API and usage

In order to start running workflows, the first thing client applications should do is create their implementation of the *WorkflowTaskDispatcher*, as described in section 4.1.4. After that, they can

instantiate a dispatcher and use it to instantiate a *WorkflowManager* as shown in listing 4.2.

```
  val myDispatcher = new MyWorkflowTaskDispatcher()

  val workflowManager = new WorkflowManager(myDispatcher)
```

Listing 4.2: Creating a workflow manager

With these created, the next step should be to create the tasks and then add the intended transitions to each task. The constructor is presented in listing 4.3

```
WorkflowTask(
    id: String,
    name: String,
    grouping: Boolean = false,
    parameters: HashMap[String, String]
      = new HashMap[String, String](),
    groupingKeys: Seq[String] = Seq()
)
```

Listing 4.3: Constructor for the *WorkflowTask* class

Then, to add transitions, the *WorkflowTask* class provides a *addTransition* method, which receives another task (which will be the destination) and an expected result (which can be of any type). The code in listing 4.4 creates two tasks and a transition from one to the other.

```
val t1 = new WorkflowTask("example-task-1", "foo")
val t2 = new WorkflowTask("example-task-2", "bar")

//Create a transition from t1 to t2, with true as the expected result
t1.addTransition(t2, true)
```

Listing 4.4: Creating a workflow

Finally, a workflow can be created by giving it a name, a collection with the tasks (the method accepts *Scala's Traversable*) and (optionally) an *HashMap* with starting values for the workflow parameters. The workflow manager object provides a *start* method with a given name that starts the workflow execution.

```
val name = "example-workflow"
val startingParameters = HashMap[String, String]("param1" -> "value1")
val tasks = Array(t1, t2)

workflowManager.createWorkflow(name, tasks, startingParameters)

workflowManager.start("example-workflow")
```

Listing 4.5: Starting a workflow

4.3 Validation and testing

The tests performed on the workflow orchestrator were essentially based on creating workflows and validating their execution order, attempting to cover the most common usage cases. Tests were performed by creating a workflow and a dispatcher with a map, matching the id of a task to a result. This way, the dispatcher can be used to manipulate the results of the task in each execution. The execution was then monitored to verify if tasks were executed and marked unreachable according to the expected output. Finally, a test with several workflows with grouping tasks was conducted to verify that execution of such tasks is also done properly. Each test will be described by first presenting a graph representation of the tested workflow, followed by the order of expected events in each execution. For each workflow there may be more than one execution, since there can be multiple paths in the same workflow, which need to be validated. For better readability, all tasks used to test will return an integer value.

The relevant events for these tests are the following:

- start(<workflow-name>): the workflow with this particular name has started;
- d(<tasks>): tasks were dispatched to execute;
- u(<tasks>): tasks were marked as unreachable;
- f(<task>, <result>): task finished with a certain result;

The inputs are events triggered by the testing agent (starting workflows, finishing tasks), while the outputs are events happening inside the workflow orchestrator itself (dispatching tasks, marking tasks as unreachable). The testing agent contained an implementation of a task dispatcher, which would have predetermined results for each task, thus allowing the simulation of the desired events as the workflow progresses. On some cases, multiple events may appear as a single input, meaning they are all required to happen in order to obtain the output, but their order does not matter. Multiple output events can also occur, meaning that they were triggered by the same input events (e.g. a task finishing may trigger some tasks tasks to be dispatched and others to become unreachable).

4.3.1 Test scenario 1: sequentiality and parallelization

The purpose of this first test is to test both sequentiality and parallelization. To achieve this, the workflow shown in figure 4.4 was designed, in which task *t2* can only run after task *t1* and tasks *t3* and *t4* can run at the same time. The results of this test are show in table 4.1.

4.3.2 Test scenario 2: alternatives

The main goal of this test is to test alternative routing, so the workflow shown in figure 4.5 was made to have two distinct paths from task *t1*. In order to test both scenarios, a test for each possible path must be made, as shown in tables 4.2 and 4.3.

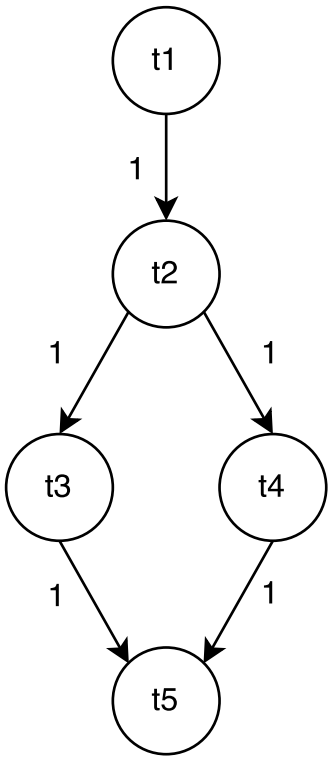

Figure 4.4: Workflow used for test scenario 1

4.3.3 Test scenario 3: grouping

To properly test grouping, the scenario should have multiple workflows running simultaneously, while contemplating the possibilities of the groupable tasks becoming either ready or unreachable. To achieve this, the workflow shown in figure 4.5 was used again, but grouping was activated for task *t3*. Three workflows with this structure were used, with two of them following the input events described in table 4.2 (executing *t3*) and the other one with the input events described in table 4.3 (making *t3* unreachable).

Since the used method of description for the other test executions may become quite complex when describing three simultaneous workflows, a more descriptive representation of the test results is more understandable:

1. The first two workflows did not execute *t3* immediately after *t2* was finished, but rather when it was ready or unreachable in all workflows;
2. Although the third workflow was never able to execute *t3*, it properly warned the grouping actor that it was unreachable, thus not stopping the group from being executed;
3. *t3* was only executed once. After that, *t4* was properly executed by the first two workflows.

This shows that grouping behaves according to what is expected by the requirements of the system.

Table 4.1: Test scenario 1

Inputs	**Output**	**Comments**
start(wf1)	*d({t1})*	
f(t1, 1)	*d({t2})*	
f(t2, 1)	*d({t3, t4})*	t3 and t4 can run at the same time
f(t3, 1); f(t4,1)	*d({t5})*	
f(t5, 1)		

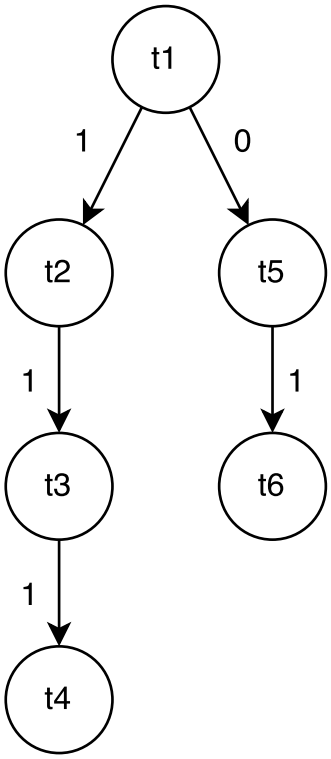

Figure 4.5: Workflow used for test scenarios 2 and 3

Table 4.2: First possible path of test scenario 2

Inputs	**Output**	**Comments**
start(wf2)	*d({t1})*	
f(t1, 1)	*d({t2}); u({t5, t6})*	
f(t2, 1)	*d({t3})*	
f(t3, 1)	*d({t4})*	
f(t4, 1)		

Table 4.3: Second possible path of test scenario 2

Inputs	**Output**	**Comments**
start(wf2)	*d({t1})*	
f(t1, 0)	*d({t5}); u({t2, t3, t4})*	
f(t5, 1)	*d({t6})*	
f(t6, 1)		

Chapter 5

Task locking and prioritization: *NoQueue*

NoQueue , as in "*No*t only a *Queue*" is a solution for handling the execution order of tasks, assigning them to an arbitrary number of consumers while doing what is possible to dispatch these tasks in the priority they were given and guaranteeing that the maximum number of consumers are busy. Each task may also request locks, imposing some rules on tasks that can run simultaneously. Furthermore, tasks can be distributed among different queues, so that consumers can selectively choose what tasks they want to run, as long as producers organize them in a commonly agreed manner.

A task is basically a structured message containing the following fields:

Subject: some meaningful string that a worker knows how to interpret and can be used to identify the nature of the task somehow;

Body: raw bytes that contain all the information the workers need to run the task. It can be viewed as a payload which is ultimately, along with the subject, what needs to be delivered to a worker so it can perform the task. *NoQueue* does not need to interpret this, only to ensure it arrives in its integrity to the worker it is assigned to;

Priority: an integer value which identifies the urgency of a task – tasks with higher priority should be assigned first;

Lock keys: string identifiers that represent some resource or entity the task needs to access. Each task can have many lock keys, which are divided in two types:

Exclusive locks: no other task with an equal lock key can run simultaneously with this task;

Shared locks: no other task with an equal exclusive lock can run simultaneously with this task;

Queue: the identifier of the queue the task should be sent to. Workers can then choose to run tasks of a specific queue. If this is not specified, the task simply goes to a default generic queue.

This way, much like a regular message queue, the system will have consumers sending messages to the system (incoming tasks) and consumers that receive and interpret the messages (workers that can execute tasks). However, it will additionally receive feedback from the consumers about the success and eventual result of a task, allowing it to determine which tasks are now available to run because they are no longer locked by other tasks and to store the result of the task. This allows the original producers (or other external agents) to later retrieve the result of the task they requested and act upon that.

In summary, viewed by an external agent (producer, consumer, monitor), the life-cycle of a task inside *NoQueue* is as follows:

1. A producer will create a task and send it to *NoQueue* , which will store it along with the other tasks waiting to run.
2. Eventually, a consumer will request a task to *NoQueue* , receiving in turn the higher priority task able to run at that moment.
3. The consumer should execute the task and notify *NoQueue* of its success or failure as well as sending a payload with the result of the task (if there is any).
4. All the data about the task will be kept, so that its status can be consulted. If a producer wishes to retrieve the result of a task, it can request that information.
5. After some (configurable) period of time, the task will be deleted and its information lost.

Contrarily to other modules in the rest of the solution, *NoQueue* runs as a standalone service, providing an HTTP API for interoperability with the other modules.

5.1 Design and implementation

The system is composed by a *Queue Handler* which holds all the queues and essentially performs three tasks. The first one is routing incoming requests from the interoperability layer to the appropriate queue. A default queue will always exist and will be the one used if no queue is specified. Each queue is identified by an unique string with “default” being reserved. However, other queues can be created and destroyed at will, when requested to do so. The other purpose of this handler is to hold the semaphore pool which is shared by all queues to maintain the resource locks, since these should be considered system-wide and are independent of which queue the task was inserted into. Finally, it also attributes an unique identifier to each task, which is returned to producers so they can identify the task when querying its status. These objects and their interactions are shown in figure 5.1.

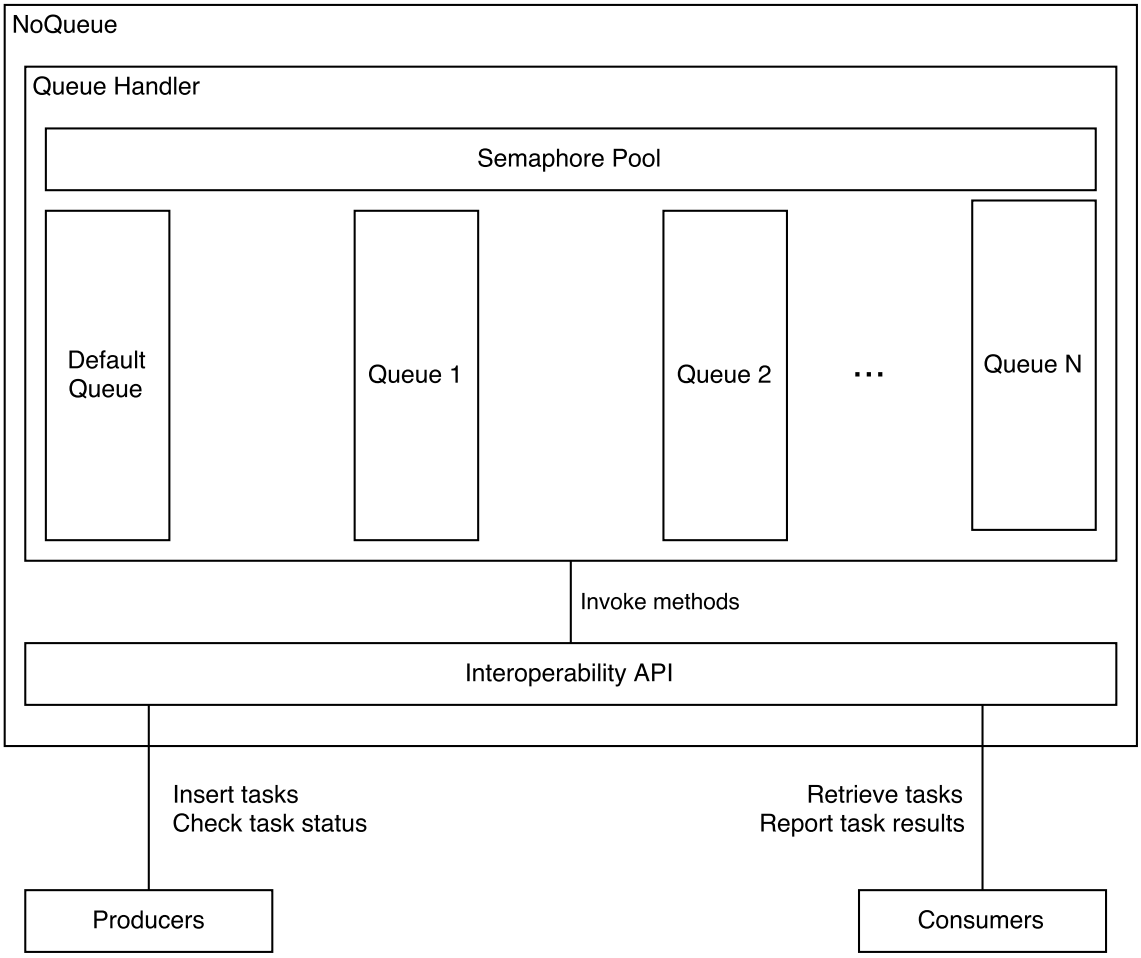

Figure 5.1: *NoQueue* objects and interactions

This section is split into three parts. First the implementation of a queue will be described by going through each of the requests it must fulfill. Afterwards, the semaphore pool is described in detail and finally, the fault tolerance contingencies are explained.

5.1.1 How a queue works

5.1.1.1 Add task

When a producer inserts a new task into a queue it will initially be placed in a cache, which the queue will periodically retrieve tasks from. This is done for two reasons: first it allows for a faster response to the producer (the only operation is an insert into a collection) and second it allows tasks to be inserted in bulk instead of one by one. Once this cache is checked, all the tasks will be inserted into an internal storage of each queue and into the *Candidates* collection. This collection is a priority queue (sorted first by the priority of the tasks and second by arrival order) that holds all tasks that are a candidate to be ready to run. Note that they are considered only a candidate,

and not ready to run, as we haven't performed any lock checks yet. This is an optimistic approach, as we are considering initially that every task should be ready to run.

5.1.1.2 Retrieve task to run

When a consumer requests a task to run, the queue will first peek the first element of the *Candidates* priority queue. Then it will check with the semaphore pool if this task is able to run, or if it is locked by other tasks with conflicting locks. If the task is not locked, the semaphore pool will automatically create the locks for the task, so immediately if a task requests the same locks, it won't be allowed to run. Since the semaphore pool has synchronized access, checking and acquiring locks in an atomic operation ensures that no conflicting tasks will be allowed to run.

Now there are two possible scenarios. If the task is currently locked, it will be moved to the *Locked* collection. This process returns to the beginning, now checking the next task on the *Candidates* queue. When a task that is ready to run is found, it will instead be moved to the *Running* collection and its information will be sent to the producer that requested it. If, however, the *Candidates* collection eventually is emptied, there are no tasks available to run and the consumer will be notified of this.

5.1.1.3 Task complete

When a consumer notifies *NoQueue* that it has completed a task (with success or not), it will send back two payloads. One contains a response containing some sort of result of this task. The other contains information about information about the cause of error, if it happened. The responsibility of interpreting this information is of both the producers and consumers, not of *NoQueue* . These notifications are also placed on a cache, just like incoming tasks, so that the incoming requests can be replied to as quickly as possible. When the notification is answered, the task will be removed from the *Running* collection and the semaphore pool will be asked to release the locks related to that task. It is now the semaphore pool's responsibility to determine what tasks could now potentially be able to run and warn the queue handler of this, which will request their respective queues to move the tasks from the *Locked* collection back to the *Candidates* collection.

5.1.2 Semaphore pool

Since resource locks are shared between all queues, an isolated component was created to keep track of resource locks. So, the semaphore pool was created, according to the class diagram in figure 5.2.

The name comes from the fact that, internally, this object will contain a map of resource identifiers to semaphores. Each semaphore keeps track if the resource has an exclusive lock or not (since it can only have one at any given time), how many shared locks it has and a list of tasks that were denied access to that particular resource, called the semaphore's waiting list. A semaphore is considered clear and it's resource available if it has no locks whatsoever. When this happens the semaphore is removed from memory to avoid growing indefinitely. This waiting list exists so

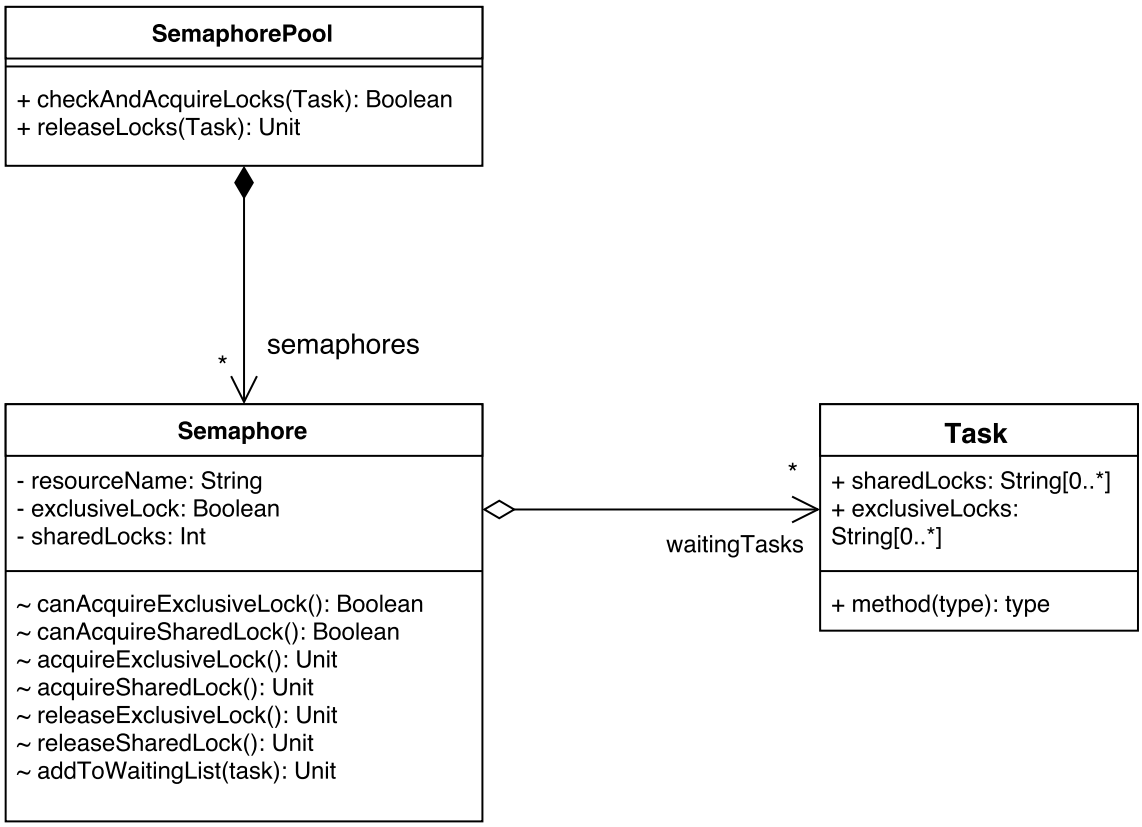

Figure 5.2: UML class diagram of a semaphore pool

that when a resource is free, we can selectively unlock the tasks that requested that lock. Note that this does not guarantee that the task is immediately ready to run, because the pool will only add the task to the waiting list of the first semaphore that locked the task. This is a balanced solution because on one hand, it doesn't have to consider all tasks as *candidates* again every time another task ends and, on the other hand, doesn't need to perform as many checks to determine if a task is locked or not, stopping immediately after finding a single resource that is locked. Note that access to this semaphore pool is synchronized, since the various queues could attempt concurrent access, leading to race conditions.

To ensure that locking is performed properly, the semaphore pool must provide queues with a way to simultaneously check if a task is locked and acquire the locks if it can do so. Otherwise, the following scenario could occur:

1. Task *A* and task *B* would simultaneously attempt to access the semaphore pool to request an exclusive lock on resource *r*.
2. Task *A* would be granted access and receive permission to run.
3. Task *B* would be immediately granted access, also receiving permission to run.
4. Now task *A* is given access to lock *r*.

5. Task *B* now tries to lock *r* but is already locked.

However, if the check and acquisition of a resource is done in a single operation, the same scenario plays out as follows:

1. Task *A* would be granted access and receive permission to run, immediately locking *r*.
2. Task *B* now receives access to the semaphore pool, and would be denied locking *r*.
3. Task *B* is successfully denied to run and will only be able to do so when task *A* is finished.

As such, the semaphore pool provides a single method to check and acquire locks, which works as described in algorithm 1.

Algorithm 1 Semaphore lock acquisition

```
if exists a semaphore for any of the exclusive locks then
    add task to semaphore waiting list
    return false
else
    if exists a semaphore for any of the shared locks with an exclusive lock then
        add task to semaphore waiting list
        return false
    else
        acquire locks
        return true
    end if
end if
```

Acquiring a lock first means finding a semaphore for a given resource and creating it if it doesn't exist. Then, either the exclusive lock value is set to *true* or the shared lock counter is incremented. When a task is complete, the semaphore pool offers a method to release all locks held by that task, which simply goes trough all of the task locks and changes their values accordingly. When any lock is removed from a semaphore (either inclusive or exclusively) the pool will check if it is now empty and removes it from memory if it is.

5.1.3 Fault tolerance

In order to improve the reliability of *NoQueue* , some sort of fault tolerance needed to be implemented. In case of a catastrophic failure where the system would go down (e.g. an unhandled exception or the physical machine itself going down), it could cost the system a lot of processing time. If this happened and *NoQueue* had no way to recover its internal state upon being started again, the producers and consumers would have no way to communicate with each other about a task that was already running. So the producers would have to send the task again and the processing of the previous task would be done again. Since in the system this solution will integrate with there could be very time consuming tasks, this could generate a lot of waste.

```
val noQClient = new NoQClient("http://localhost:8086")
```

Listing 5.1: Instantiating a *NoQClient*

To avoid this, an internal state recovery system was implemented, which essentially logs to a file every event that occurs in the system. When *NoQueue* starts, it reads this file and every event is recreated in the same order, thus restoring the internal state of the system.

This was achieved using the *Observer* pattern. The queue handler was extended to support observers to be attached to it. Every time a relevant event happens, the observers are notified. An observer was then implemented in order to register the event log, as well as providing a method to read back the log so the events can be recreated upon system start.

5.2 API and usage

As already mentioned, *NoQueue* provides an HTTP API so that producers and consumers can interact with it. Additionally a *NoQueue* client library written in *Java* is provided that provides methods that properly perform the HTTP requests, interpret and return their response. What follows are examples on how the most common requests can be performed using this library.

Setting up the client After including the library into the client project, the first step is to create an instance of the *NoQClient* class, which only needs the address where *NoQueue* is running, like shown on listing 5.1.

This constructor will automatically perform a test request and will throw an exception if *NoQueue* can't be reached through the provided address.

Producer sending a task Usually, a producer will want to send a task to *NoQueue* and eventually retrieve its result. So first, it can use the provided *addTask* method and then periodically invoke the *taskStatus* method using the identifier given by the first method. This second method will return a *StatusReport* object, containing everything that was sent by the consumer that executed the task (the response and exception payloads) as well as a string that identifies the state of the task. Note that some parameters are optional and were omitted in the following example for the sake of simplicity.

Consumer requesting a task On the other side, the consumer can use the *requestTask* method, which only requires a consumer identifier string. It should be the responsibility of the client application to guarantee that these identifiers are unique across every consumer. Then, it should interpret the subject and payload of the task and do whatever computations arise from that. Finally, it can use the *taskDone* method to send the result back to *NoQueue* .

The client provides access to every endpoint of the HTTP API, but for the purpose of this document these examples provide enough insight about the main goals and common usages of the

```
val subject = "task1"
val payload = "some␣payload␣content"
val priority = 1

val taskId = noQClient.addTask(subject, payload, 1)

//Now we can periodically get a status report until the task is done
var statusReport = noQClient.taskStatus(taskId)

while(statusReport.state != NoQTask.State.COMPLETED
    || statusReport.state != NoQTask.State.FAILED) {
  statusReport = noQClient.taskStatus(taskId)
  Thread.sleep(1000)
}
```

Listing 5.2: Adding a task to *NoQueue* and checking its status

```
val myConsumerId = "worker1"
val task = requestTask(myConsumerId)

val result = executeTask(task.subject, task.payload)

noQClient.taskDone(myConsumerId, task.id, result)
//or if the task fails
noQClient.taskFailed(myConsumerId, task.id, result, "error")
```

Listing 5.3: Requesting a task and sending feedback about its execution

client. Client applications can refer to the source code of the client which contains meaningful comments for each method.

5.3 Validation and testing

Although throughout development the various components of this system were tested separately, what will be described here are the end to end tests to the system as a whole. To achieve this, tasks that were inserted, executed and completed were controlled by the test itself, validating that the tasks handed to consumers were done so in the expected order. The test will act as both a consumer and producer with infinite capacity (i.e. can wait on and execute any number of tasks simultaneously). The main purpose of these tests is to ensure that priority and locks are ensured, which are the main responsibilities of *NoQueue*, so the remaining fields were given irrelevant values that are not described on this document for having no relevance to the tests. Each test will be described as a sequence of events, which can be one of the following:

- *ins(<task>, <priority>, <exclusive locks>, <shared locks>)*: a task was inserted into the system.
- *done(<task>)*: a task was completed and the system was notified.
- *request*: a task was retrieved, obtained either the identifier of the task or none if there are no tasks to run. The purpose of the tests will be to validate the output of this particular event.

5.3.1 Testing lock combinations

The tests described in tables 5.1, 5.2 and 5.3 validate if tasks are properly locked by testing all the different resource lock combinations.

Table 5.1: Testing exclusive locks

Events	**Output**	**Comments**
ins(t1, 1, {r1}, {})	-	
ins(t2, 1, {r1}, {})	-	
request	t1	t1 arrived first
request	{}	Both tasks requested the same exclusive lock
done(t1)	-	
request	t2	

Table 5.2: Testing shared locks

Events	**Output**	**Comments**
ins(t1, 1, {}, {r1})	-	
ins(t2, 1, {}, {r1})	-	
request	t1	t1 arrived first
request	t2	Shared locks should not lock each other

Table 5.3: Testing exclusive locks against shared locks

Events	**Output**	**Comments**
ins(t1, 1, {r1}, {})	-	
ins(t2, 1, {}, {r1})	-	
request	t1	t1 arrived first
request	{}	Shared lock is incompatible with exclusive lock
done(t1)	-	
request	t2	
ins(t3, 1, {}, {r1})	-	
ins(t4, 1, {r1}, {})	-	
request	t3	t3 arrived first
request	{}	Exclusive lock is incompatible with shared lock
done(t3)	-	
request	t4	

5.3.2 Testing priorities

The test described in table 5.4 validates if tasks are dispatched according to the priority they were assigned.

Table 5.4: Testing if tasks come out with decreasing priority

Events	**Output**	**Comments**
ins(t1, 1, {}, {})	-	
ins(t2, 5, {}, {})	-	
ins(t3, 3, {}, {})	-	
ins(t4, 7, {}, {})	-	
ins(t5, 1, {}, {})	-	
request	t4	
request	t2	
request	t3	
request	t1	t1 and t5 have the same priority, but t1 was inserted first
request	t5	

Chapter 6

Workers engine

The workers engine aims to provide a library for the *JVM*, written in *Scala* to abstract the execution and retrieval of tasks. It allows the client application to create configurable Worker Pools composed by a single manager and a parameterizable number of workers. These workers will continuously retrieve tasks to execute using a provided task retriever, which should implement the logic needed to obtain a single task. The tasks provided by the retriever should implement a *run* method which the worker can call to perform the desired work. Finally, the worker should report the success or failure and the result of a task. The logic of this report sending should also be implemented by the retriever. In summary, the workers engine is only responsible for the management of workers (including fault tolerance) and the continuous retrieval of tasks and execution by these workers. The way a single task is retrieved and how the outcome is reported should be left to whoever is using the library.

The actors model is very interesting for this implementation for various reasons:

1. *Akka* supports distributed actor systems, so it would be easy to deploy this engine in different machines;

2. Supervising the workers and implementing contingency methods is made much easier by *Akka* supervision;

3. Conceptually, workers are very similar to actors, with the only difference that in this scenario they will block while processing messages, as they will be processing the messages themselves;

4. Thread management is taken care of by *Akka* itself, so it is one less concern while implementing the engine;

6.1 Design and implementation

6.1.1 Workers and the task retriever

A worker is an actor that continually attempts to retrieve tasks to perform. Once a task is retrieved, the worker first warns its supervisor about the task it received and the executes the task. To retrieve tasks, the actors are given an implementation of a task retriever. The basic principle of a task retriever is that it fetches information from some source (e.g. a file, database or in this particular case, *NoQueue's* HTTP API) and uses that information to create a *WorkerTask* object, with a proper *run* method that does whatever it should do taking into account the information in the task itself. This way, upon receiving the task object, the worker simply invokes the *run* method, so the object itself should contain everything it needs to properly execute the desired task. Thus, it would be fair to compare a *WorkerTask* with a *Java Runnable* and the worker acting as *Thread*.

To create a task retriever for the workers to use, the user should create a class that extends the *TaskRetriever* abstract class, implementing the methods show in listing 6.1.

The worker actor accepts three different messages: *Next*, *Pause* and *Unpause*, and has four possible states: *Receiving*, *TaskDone*, *Paused* and *PausedTaskDone*. A worker always starts in the *Receiving* state and will send a *Next* message to himself to start polling the retriever upon instantiation. While in this state, the worker will respond to a *Next* message by attempting to obtain and execute a task from the retriever. If no task is found, the worker wait a period of time until trying again, increasing that time exponentially upon each failure, to avoid unnecessary requests to the retriever [SM03]. However, if a task is found, the worker will first warn its manager that it is executing that particular task, as in case of failure it will be the manager's duty to notify the task retriever. If such a failure happens, the worker will throw a *WorkerFailureException*, containing the exception thrown by the *run* method of a task, the worker's unique identifier and the task that failed. This is so that the manager can handle the failure through its supervision strategy. After that, the worker will invoke the *run* method of the task object and, upon no exception thrown, transition to the *TaskDone* state and send itself a *Next* message. Upon receiving the *Next* in the *TaskDone* state, the worker will notify both the retriever and supervisor of its successful completion of the task. Finally, it will transition back to the *Receiving* state and send a *Next* message to itself, resuming the task retrieval loop. These states and transitions are graphically represented in figure 6.1[spe17].

```
abstract class TaskRetriever(parameters: HashMap[String, Any],
                             val workerLogger: WorkerLogger) {
  /**
    * Initialize the retriever
    * (e.g. create database connection, test HTTP endpoint)
    * @return whether the initialization was successful or not.
    */
  def initialize(): Boolean

  /**
    * Check if the retriever is available
    * (e.g. the HTTP server is up, database connection is OK).
    * @return whether the retriever is still available
    */
  def getStatus(): Boolean

  /**
    * Retrieve a task.
    * Should throw a NoTaskRetrievedException if there are
    * no available tasks at that moment.
    * @param workerId the identifier of the worker.
    *                 Can be useful to perform some validations.
    * @return a task for the worker to run.
    */
  def retrieve(workerId: String): WorkerTask

  /**
    * Notify the retriever that a task was successfully completed.
    * @param workerId the identifier of the worker.
    *                 Can be useful to perform some validations.
    * @param task the task that the worker executed.
    */
  def done(workerId: String, task: WorkerTask): Unit

  /**
    * Notify the retriever that a task has been terminated
    * with an error.
    * @param workerFailureException a wrapper for the exception
    * generated by the worker, containing the worker id,
    * the task that went wrong and the
    * exception thrown during its execution.
    */
  def failed(exception: WorkerFailureException): Unit
}
```

Listing 6.1: The *TaskRetriever* abstract class declaration

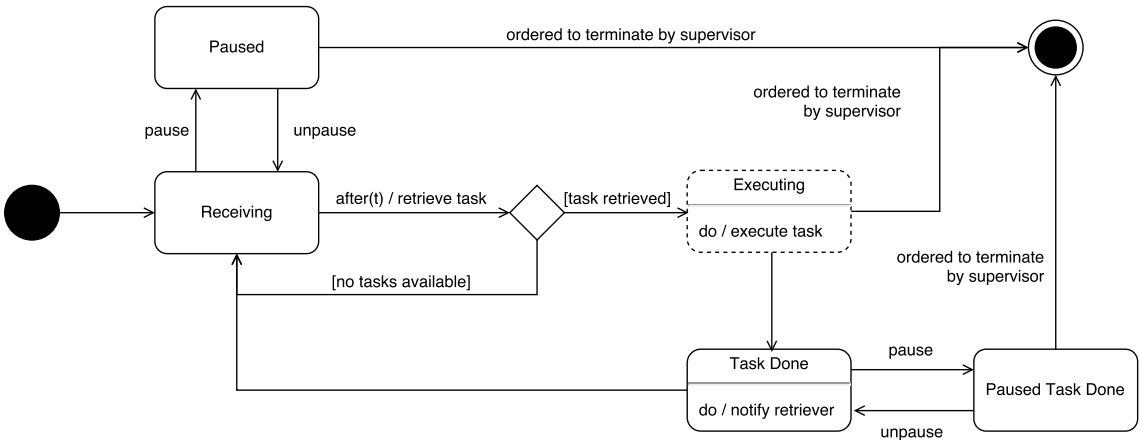

Figure 6.1: A UML state machine representing the states of the worker. The *Executing* state appears dashed because the actor does not change behavior.

Please note that the actor can be paused and unpaused during both *Receiving* and *TaskDone* states, transitioning to *Paused* and *PauseTaskDone*, respectively. This will be used by the manager as explained below.

6.1.2 Worker manager

All worker actors are supervised by a single manager actor that ensures that the workers and retriever are running properly and that failures are properly reported to the retriever, since upon a catastrophic failure of a worker, it can not be relied upon to do so.

Upon creating a worker pool, the manager is immediately created, but it will stay idle until the pool receives order to start. When this happens, the first thing the manager will do is to initialize the task retriever, by invoking its *init* method. If this succeeds, a number of worker actors will be created according to the configuration received by the worker pool. While the workers are running and performing tasks, the supervisor will be notified by an actor when he starts or completes a task, keeping an internal record of this information. It keeps this information, because when an actor throws a *WorkerFailureException*, the manager will first notify the retriever about this failure and then restart the actor so that it can continue retrieving tasks without any problems caused by any eventual internal state corruption.

Additionally, upon failure to communicate with the retriever, a worker will throw a proper exception. The manager's supervision strategy dictates that when this happens a contingency measure should also be applied. The main purpose is to avoid that all workers keep trying to communicate with the receiver while it is unavailable, since in some scenarios this can even make it harder for the retriever to come back up (e.g. an HTTP server that is receiving too many requests). To prevent this, the manager first pauses all the workers by sending them a *Pause* message. This makes them switch to the *Paused* or *PausedTaskDone* accordingly, where they will ignore all *Next* messages until receiving an *Unpause* message, where they will resume their normal functioning by returning to their previous state and sending a *Next* message to themselves. These distinct states

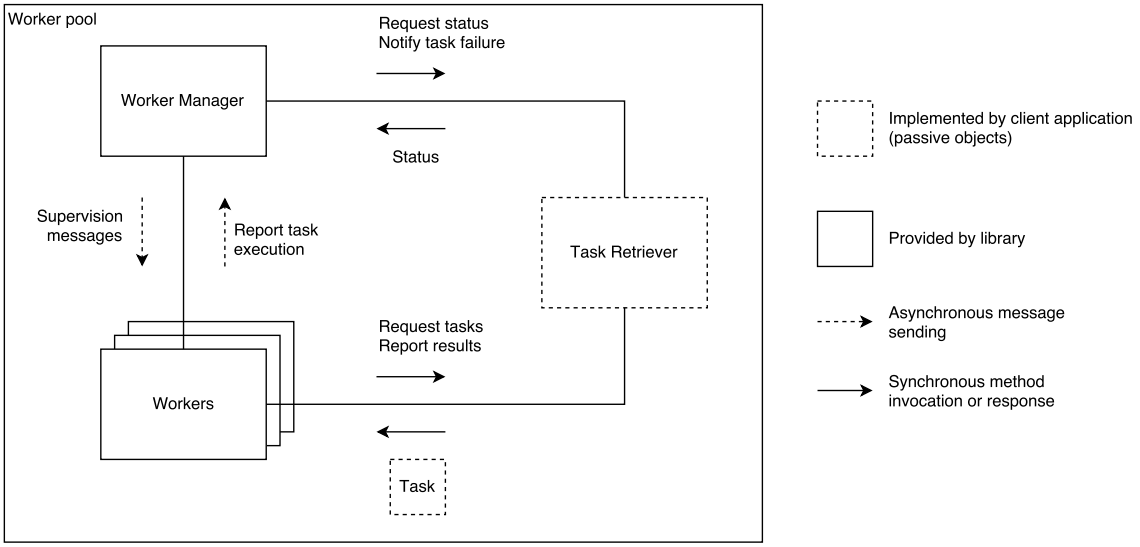

Figure 6.2: The components of a worker pool and how they interact

are important, because when a communication failure with the retriever fails, the worker could be attempting to retrieve a task (*Receiving* state) or communicating a task completion (*TaskDone* state). When unpaused, it is crucial that the worker returns to the same state so it can resume exactly what it was doing, and no information is lost.

After pausing all workers, the manager will periodically check if the retriever is available again, using the *getStatus* method. Once it is back up it will unpause all workers, which will resume retrieving tasks or notifying about complete tasks. This way, a single periodic request will be made to the retriever and avoid wasting various workers trying and failing to communicate with the task retriever, wasting resources and possibly contributing to the retriever's cause of unavailability.

6.1.3 Worker pool

A worker pool is composed by a task retriever, a single worker manager and an arbitrary number of workers, as shown in figure 6.2, which interact as described during this chapter.

Besides the customization provided by the implementation of the retriever and task execution, there are a set of options the worker pool can receive through a configuration object, which should specify the following parameters:

Number of workers the amount of workers that should be created.

Workers maximum poll time the maximum amount of time a worker should wait before attempting to retrieve a new task.

Workers base poll time the base poll time used by the worker (i.e. the minimum waiting time without the back-off and random interval)

```
class WorkerPool(name: String,
                 workerPoolConfig: WorkerPoolConfig,
                 taskRetriever: TaskRetriever,
                 workerLogger: WorkerLogger = new DefaultWorkerLogger)
```

Listing 6.2: *WorkerPool* constructor

Task retriever reconnect maximum poll time the maximum amount of time the supervisor will wait before attempting to reconnect with the task retriever.

Finally, the user can specify their own logging logic, by extending the *WorkerLogger* class, customizing how each level of logging is handled (error, info, warning and debug) and passing an object of that class to the logger. However, a default logger is provided that simply prints the logs to the console.

After being instantiated, the user should call the *start* method of the worker pool, so the manager can create the actors which will start retrieving and executing tasks.

6.2 API and usage

The workers engine is compiled into a *jar* file, which other *JVM-based* projects can import. In order to make use of it, client applications must instantiate a *WorkerPool* object, which has the constructor shown in listing 6.2.

The first parameter is the *name* of the worker pool, which should be unique, as it will be used to create the actor system that will support the structure described in section 6.1.3. It should be noted that the client application may create any number of *WorkerPool* objects, which will work completely independently (provided that they each have a unique name, which should be the responsibility of the client). Afterwards, the *WorkerPoolConfig* is an object containing values for the parameters described in 6.1.3 and a task retriever implementation as presented in section 6.1.1. The final parameter is a *WorkerLogger* object, which is optional and will default to an instance of *DefaultWorkerLogger*. The purpose of this object is to allow the client application to customize how the various output messages produced by the worker pool are handled. To do this, the methods in listing 6.3 need to be implemented.

Each method represents a different log type, and receives both the message that should be printed, but can also receive a *Throwable* object that could eventually originate from the execution (e.g. logging an exception). The default logger provided by the workers engine simply logs all messages to the system console with a prefix containing the log type. If a throwable exists, its stack trace will also be printed.

```
abstract class WorkerLogger {
  def info(message: Any, throwable: Throwable = null)
  def error(message: Any, throwable: Throwable = null)
  def warn(message: Any, throwable: Throwable = null)
  def debug(message: Any, throwable: Throwable = null)
}
```

Listing 6.3: *WorkerLogger* class declaration

6.3 Validation and testing

The tests envisioned for the workers served two purposes: validate if the workers are capable of correctly retrieving and executing tasks and measure their performance in terms of task completion time. This second part is important in order to assure that the system is not losing too much performance compared to just creating threads simulating the logic of a worker.

To achieve this a task retriever for the sole purpose of testing was created. This retriever starts out with a configurable number of tasks to distribute. Each task just asks the calling thread to sleep for 1000 milliseconds and terminates. The retriever will hand out tasks to any worker that requests it, until it has no more tasks. Once this happens, it will output the start and end time of each task to a file so we can analyze the data.

With this scenario set up, the theoretical total completion time (T) depends on:

- T_{task}: the completion time of a single task;
- N_{tasks}: the total number of tasks;
- $N_{threads}$: the number of threads available to assign to workers;
- $N_{workers}$: the number of workers available;

and is given by:

$$T = T_{task} \times \lceil \frac{N_{tasks}}{min(N_{threads}, N_{workers})} \rceil$$

The actor system is using the *Akka* default dispatcher, whose default configuration dictates that the number of available threads should be three times the number of available processors. Since the tests were running on a machine with 4 processors, the actor system should have 12 threads available. However, the workers are not the only actors in the system, since the actor manager also processes messages frequently because workers will be notifying it about tasks they are running. Therefore, the theoretical values were calculated with a pessimistic approach, considering the worker manager always has a thread, thus only having 11 threads available.

The first tests with an initial version showed the results presented in table 6.1. Using only 1 or 10 workers yielded results very similar to the theoretical value. However when using 20 workers, the measured values were way worse than the theoretical value. Repeated tests showed consistent

Table 6.1: Total completion times of tasks in first workers test

Workers	Tasks	Measured time (ms)	Expected time (ms)
1	100	100689	100000
10	100	10079	10000
20	100	100492	9091
1	200	200917	200000
10	200	20147	20000
20	200	108255	18182

values with only slight variations when using 1 or 10 workers. On the other hand, using 20 workers was consistently worse and the values would have high variation.

To investigate the cause, a more thorough test was done, fixing the number of tasks to 100, but testing every number of workers from 1 to 20. The results were then plotted in the graph show on figure 6.3. This was helpful, because it is possible to observe that until 11 workers, the measured and expected times remain very similar. After that, completion time appears to increase or decrease randomly, with repeated tests showing different values.

Analyzing the data more closely revealed another fact about why this was happening. It was observed that tasks were executed according to expected, the workers took a very long time executing the few final tasks. Considering data and analyzing the solution, a possible cause was immediately identified. When workers are attempting to retrieve tasks, if they receive none, they will wait for some time and then attempt to retrieve a task again. The worker would ask the calling thread to sleep for a period of time and then would send a message to itself to retrieve a task again. Note that this would not release the thread, so other workers would not be able to use it in the meantime. This made it so that when there were no more tasks to retrieve, the workers would start occupying threads for a very long time, until releasing it for another worker which already had a task to warn the retriever that the task was completed. To test if this was the cause, this waiting time was changed. Instead of using a thread sleep, the worker now sent a scheduled message to itself. *Akka* provides a way to do this, so it will only attempt to deliver the message after the requested period of time. Using this new version, the same test was executed and the results are shown on figure 6.4, which shows that values are now very close to expected across any number of workers.

Using the default dispatcher configuration, *Akka* will take away a thread from a worker to give to another every 5 messages that the worker processes. Once the retriever ran out of tasks, a lot of workers would hold on to threads a very long time because of blocking while waiting to retry. This meant that workers trying to notify retrievers that they had completed a task were not able to acquire a thread to process the message they had sent themselves.

This particular testing scenario was helpful by allowing us to detect bottlenecks in the workers engine and validating that the overhead introduced is minimal, since measured times are very close to theoretical calculations.

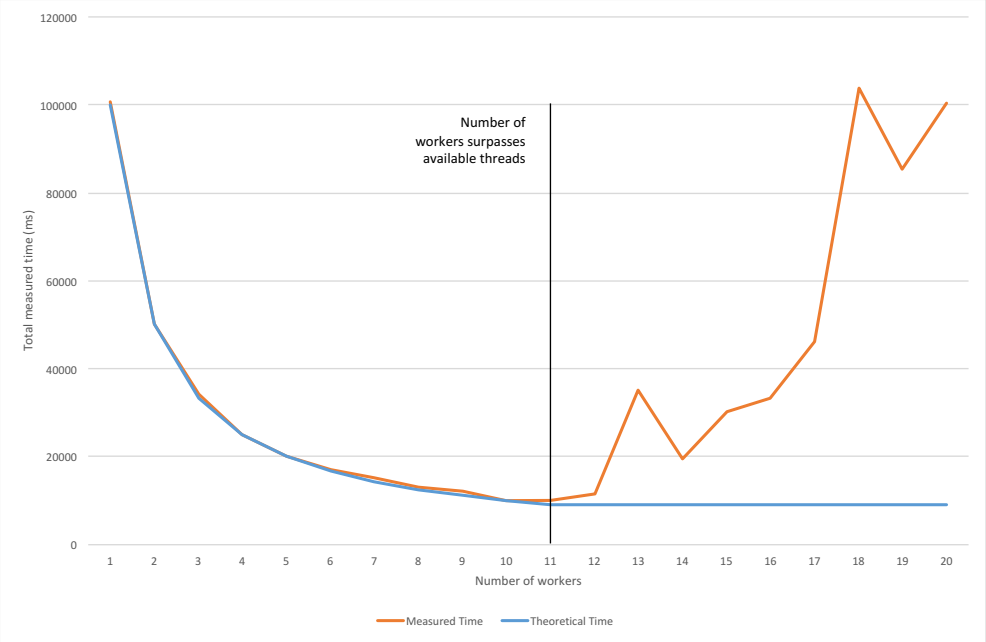

Figure 6.3: Measured and theoretical times for a varying number of workers executing 100 tasks

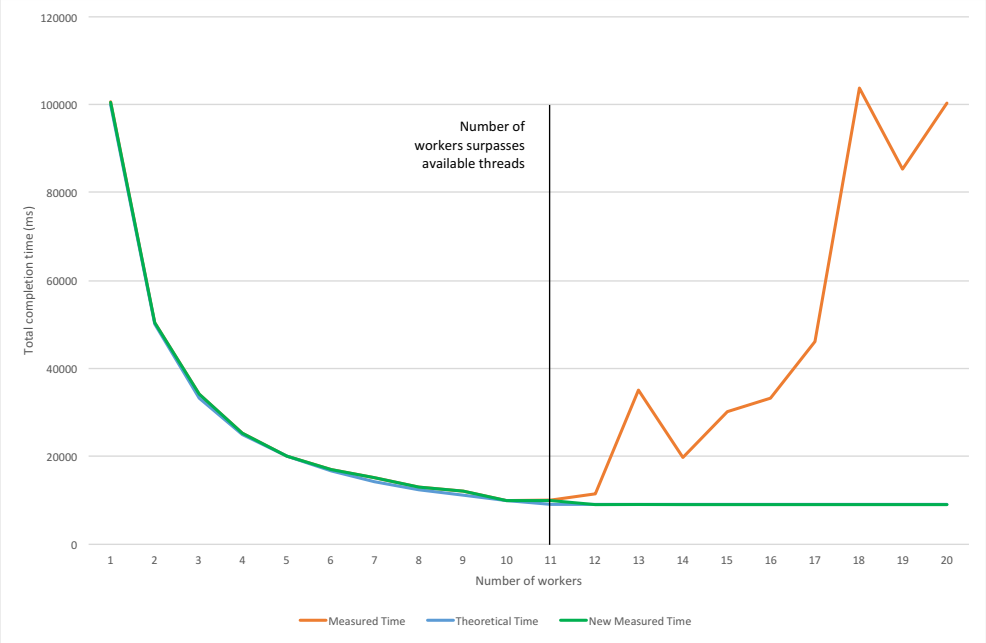

Figure 6.4: New and old version measured and theoretical times for a varying number of workers executing 100 tasks.

Chapter 7

Integration with previous system

Now that each layer of the solution has been explained in detail, it is possible to understand how the existing system can make use of them and what changes need to be made in order to support them. Essentially, the entire component that deals with workflow handling and execution must be replaced to use the workflow orchestrator instead, which in turn needs to send tasks to *NoQueue* . The workers engine will also need a retriever to fetch tasks from *NoQueue* and an implementation of tasks that knows how to execute *Colbi's* own tasks.

7.1 Workflow orchestrator

The first thing that needed to be done in order to use the workflow orchestrator was to create a converter that parsed the existing workflow files, creating workflows instances according to the new workflow model. This was done to maintain backward compatibility. However, as will be explained later, to fully utilize the new system features, the existing workflow definition model will need to be extended or completely overhauled. Since the existing tasks do not support returning results and would need to be changed to do this, it was decided that initially, the new tasks would all return a boolean value, simply stating if the task was executed without errors or not.

Taking this into consideration, and referring back to the example show on listing 1.1, the following conversion was made:

- The name of the workflow will be the old name, with the identifier of the file appended (to guarantee uniqueness);
- Initial workflow parameters will always be empty;
- The identifier of the new tasks will be the name of the old tasks, with the identifier of the file appended;
- The name of the task will be the "task" attribute of the old version, since this was what identified what the task was supposed to do;

- Parameters remain the same;
- For every task in the "success" event, a transition is added expecting *true* as a result. In the case of the "failure" event, the transition will expect *false*;
- A "grouping_parameters" attribute, containing an array of strings (which can be empty) was added to the tasks. If this attribute is present, grouping will be activated for that task, using the specified keys as grouping parameters;
- Priority and resource locking are not yet supported.
- A "channel" attribute was also added, allowing the task to be sent to a specific *NoQueue* queue. If specified, *NoQueue* will always be requested to create this queue if it does not exist.

With tasks created, it is required to implement the task dispatcher. In this case, the dispatcher sends the tasks to *NoQueue* and then periodically checks its status. Once it receives information that the task is complete or failed, it considers it done with and returns the boolean result accordingly. Tasks sent to *NoQueue* will have the following attributes:

Subject: the name of the task;

Body: a *WorkerEnvelope* object is created, which contains all information needed to execute the task. It is serialized to a *base64* string which the workers will be able to deserialize. The entire contents of this object are not relevant, since they will only interfere with *Colbi* specific logic inside the tasks. It is only relevant to know that the task's specific and workflow parameters are sent through this object.

Priority: is set to 1;

Locks: both shared and exclusive are empty;

Queue: the channel attribute of the task.

When the task is finished, the response body will contain a *WorkerEnvelope* object which will be deserialized and properly merged with the one that was sent initially. Additionally, the workflow parameters of the task are updated. If the task also contains an exception, it will be sent to a different component of *Colbi* that handles task exceptions.

NoQueue will be running as an external service, and its HTTP address is specified in a new field added to *Colbi's* configuration file.

7.2 Task execution

With the tasks sent to *NoQueue* , a service was created that uses the workers engine to fetch them. This is implemented as a secondary *main* class inside the *Colbi* project, which therefore has access

to all implemented classes, including the ones required to execute the tasks. The same *jar* file that runs *Colbi* can then be used to start the workers service, by simply choosing a different main class. To achieve this, a task retriever was implemented that made use of the *NoQueue* API to perform its operations, as follows:

Initialize : creates the *NoQClient* object for internal usage;

Retrieve task : uses the *requestTask* method to attempt to obtain a task;

Task done : uses the *taskDone* method, sending the serialized *WorkerEnvelope* as the result;

Task failed : uses the *taskFailed* method, sending the serialized *WorkerEnvelope* as the result and the serialized *WorkerFailureException* as the error;

Get status : checks if the HTTP server where *NoQueue* should be is available;

The *run* method of the tasks provided by the retriever will simply deserialize the provided *WorkerEnvelope* object and use it to run the same logic that was used in the old system. Note that an instance of workers can be configured to retrieve tasks from a specific queue, allowing workers to be specialized in performing certain tasks, as long as workflows are also created accordingly.

With this, *Colbi* is now sending tasks to *NoQueue* instead of executing them and is capable of running a service (or many instance of it) containing workers that retrieve and execute the tasks.

7.3 Validation and testing

To test the entire system, a set of files were imported into the system, using the old version and new versions of the system. On one hand, this allowed the validation of the results (by comparing the contents of the database) and compare the performance of both systems. However, in the new version, workflows were changed to group tasks that can do so.

From a set of 10 files, concurrent importations of files were made with an increasing subset of those files (i.e. importing only the first file, then the first two simultaneously, then the first three, and so on). For each subset of files, their simultaneous importation was done 100 times, in order to obtain the average time. The results are presented in figure 7.1. Although in the old system the files are processed concurrently, since there is no grouping, some tasks may compete for resources while still occupying one thread, and the amount of tasks performed is higher.

It is visible that until three simultaneous files the old system performed better, from there the new system showed lower processing times. As expected, due to the overhead introduced by the new components in the system, with a low number of simultaneous files, the performance gains are not enough to compensate. However, as the number of files increases, the new system becomes progressively better, as the gains introduced overcome the introduced overhead. This is ideal, since the common usage in *Colbi* consists of importing multiple files simultaneously, making it more valuable to increase performance in this scenario, even at the cost of losing when importing fewer files.

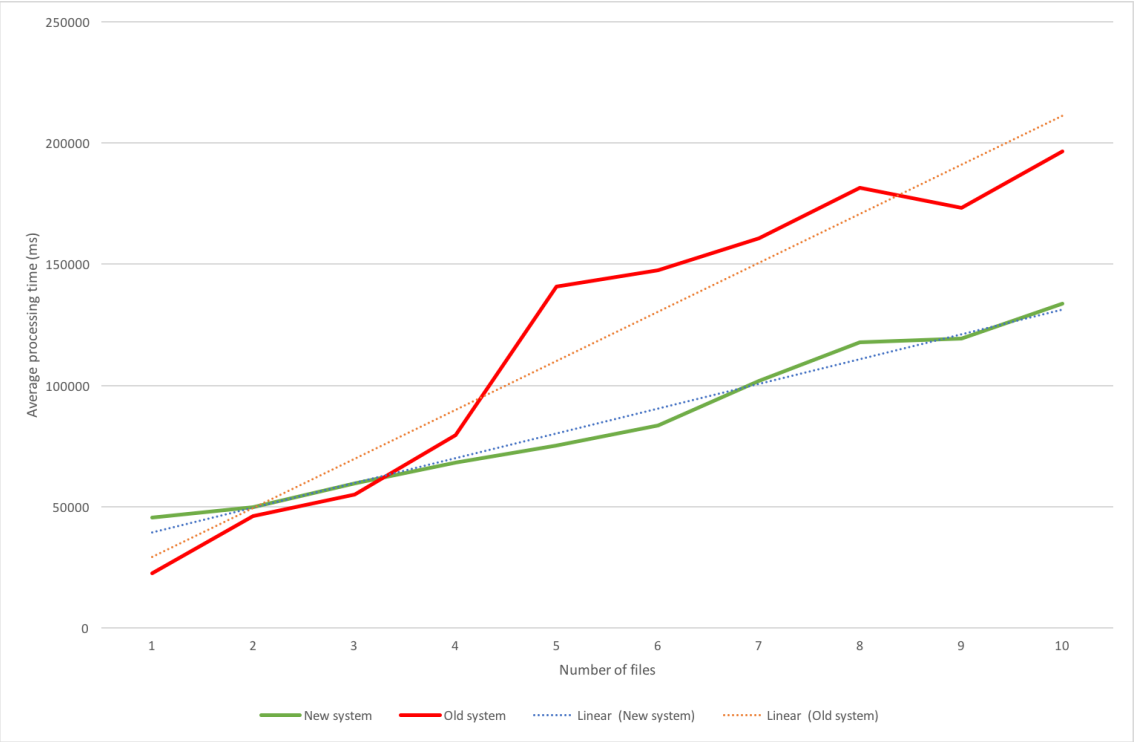

Figure 7.1: Comparison of total processing time for an increasing number of simultaneous files, in the old and new systems

Chapter 8

Conclusions and future work

8.1 Conclusions and achievements

In retrospective, the implementation of each of the modules of the proposed architecture was successful. This way, this work has created three distinct software artifacts that can be used in different scenarios and that are fully interchangeable. For example, it would be possible to replace *NoQueue* with a different agent that was able to receive and dispatch tasks, but with a different logic, integrating the workflow orchestrator and the workers engine with that new component. As a more concrete example, there are instances of *Colbi* running in production environments using *NoQueue* and the workers engine, while still using the old workflow logic. However, the regular execution logic was bypassed to instead send the tasks to *NoQueue*. Currently, the biggest of these environments contains approximately 2300 companies with more than 10000 imported files, all processed using both *NoQueue* and the workers engine. Although they do not take advantage of any of the developed scheduling rules, this setup does allow tasks to be distributed through workers spread throughout various machines and make use of the different queues to have machines specialized in running a particular set of tasks. This demonstrates the flexibility of the developed modules and the enrichment in terms of features and customization added to the workflows.

In terms of performance, it was also proven that there were performance gains when concurrently processing files. This fulfills the proposed goal, although as expected, processing a low number of concurrent files suffered a performance reduction. However, this is acceptable, as *Colbi's* usual scenarios is having several multiple files being imported simultaneously.

Unfortunately, due to time constraints not all features were integrated with the existing system, as it will have to be deeply changed to support the concepts of task prioritization, resource locking and tasks returning relevant results. However, the developed components have been validated to support these features, so a big part of the work is already done.

In general, the work developed during this thesis brought many improvements to *Colbi*, some of which are working in production environments without any major issues.

Finally, we can revisit the proposed requirements and summarize their fulfillment:

R1: Sequentiality Fulfilled and used by the final solution;

R2: Parallelization Fulfilled and used by the final solution;

R3: Alternative Implemented by the workflow orchestrator, but the existing system will need to be extended to make use of it;

R4: Grouping Fulfilled and used by the final solution;

R5: Work distribution Fulfilled and used by the final solution;

R6: Resource locking Implemented by *NoQueue*, but the existing system will need to be extended to make use of it;

R7: Prioritization Implemented by *NoQueue*, but the existing system will need to be extended to make use of it;

8.2 Future work

As already mentioned, the obvious next step in terms of future work would be to implement task prioritization and resource locking in *Colbi*. To make use of prioritization, the system itself would need to have some logic that would prioritize files according to an appropriate heuristic. On the other hand, to make use of resource locking, each task would have to be analyzed to determine what resources would be relevant to acquire a lock on. As stated initially, this would prevent tasks from blocking while waiting on a resource, thus avoiding wasting that time.

Finally, each of the developed components has some improvements that could be made, but there are a few that stand out.

HTTP may not be the optimal protocol to communicate with *NoQueue*. Other protocols could be analyzed and implemented, eventually leading to performance and reliability increases.

Instead of each worker constantly requesting tasks to the retriever, a middle-layer should be developed that requests as many tasks as workers that are available in a single message, distributing them through the available workers. This would reduce the amount of communication required, but *NoQueue* would also need to be extended to support handing out more than one task at a time.

Finally, it would be interesting to add something similar to logic gates to the workflow, with incoming and outgoing transitions. This would allow for even greater flexibility to workflows, although probably at a cost of introducing complexity to both the implementation itself and the interaction with client applications.

References

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[DVJ+15] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[fDoBR17] Guidance Note Guidance for Developers of Business and Accounting Software Concerning Tax Audit Requirements. http://www.oecd.org/tax/administration/guidancenote-guidancefordevelopersofbusinessandaccountingsoftwareconcerningtaxauditrequirements.htm, January 2017.

[Fou16] YAWL Foundation. http://www.yawlfoundation.org/, July 2016.

[GMUW08] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. Database Systems: The Complete Book. *Education*, page 1248, 2008.

[He04] Xudong He. High-Level Petri Nets—Extensions, Analysis, and Applications. pages 459–476, 2004.

[Lan16] Workflow Description Language. https://github.com/broadinstitute/wdl#workflow-description-language-wdl, July 2016.

[Mur89] Tadao Murata. Petri nets: properties, analysis and applications, 1989.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[SM03] Nah-oak Song and Leonard E Miller. On the Stability of Exponential Back-off. *Journal Of Research Of The National Institute Of Standards And Technology*, 108(4):289–297, 2003.

[spe17] OMG Unified Modeling Language specification. http://www.omg.org/spec/uml/2.5/, January 2017.

[Tav16] Apache Taverna. https://taverna.incubator.apache.org/, July 2016.

[Tri16] Triana. http://www.trianacode.org/, July 2016.

[Van98] W. M. P. Van Der Aalst. the Application of Petri Nets To Workflow Management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.

[Wil86] Robin J Wilson. *Introduction to Graph Theory*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Yam17] Yaml. http://yaml.org/spec/1.2/spec.html, January 2017.